National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

Formalizing Telephony Feature Interactions in Mizar-C

by

Stephen Rochefort

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta
Spring 1996

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10751-5

Canada

University of Alberta

Library Release Form

Name of Author: Stephen Rochefort

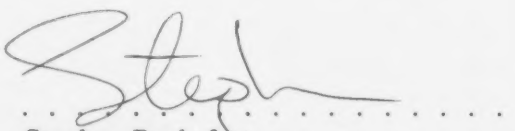
Title of Thesis: Formalizing Telephony Feature Interactions in Mizar-C

Degree: Master of Science

Year this Degree Granted: 1996

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Stephen Rochefort
4 Carmel Wynd
Sherwood Park, AB
T8A 5L1

Date: 11 December 1995

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Formalizing Telephony Feature Interactions in Mizar-C** submitted by Stephen Rochefort in partial fulfillment of the requirements for the degree of Master of Science.

Dr. H. James Hoover

Dr. B. Cockburn

Dr. J. Harms

Date: 11 December 1995

Abstract

The introduction of numerous new features into current telecommunications systems is hindered by the *feature-interaction* problem. Three specific types of feature interactions can occur. *Conflicting features* occur when two features have opposing characteristics. *Ambiguity* is caused when multiple features use the same signal for different actions. *Resource contention* occurs when multiple features require the use of a limited resource in the telephony network.

Many methods have been introduced in an attempt to detect, prevent, and resolve undesired feature interactions. Most of the effort has been spent on detecting and resolving interaction during system execution. In this thesis, a method is proposed which aims to prevent these undesirable interactions at the design stage by formally specifying features and the possible interactions. This method takes into account the fact that features are added incrementally over the life of the system. This approach ensures that no interactions occur during execution time. To limit the size of this project, this thesis demonstrates the process used to formally specify conflicting features.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to those who have helped me throughout my graduate studies.

I would like to extend my deep appreciation and admiration to my supervisor, Jim Hoover, for his support and guidance through the course of this work, and to the members of my committee for their time and evaluation of this thesis.

I extend my appreciation to Edward Ho for his brilliant discussions and explanations regarding formal methods and Mizar-C when I first started this work.

I thank my parents for their love and support, for always believing in me, and for providing me with the push in life to go after anything I desired.

Most of all, I would like to thank Rachelle who always went out of her way to provide the love and support I needed to make it through the good and bad times.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Formalizing Telephony Features . . . 2
- 1.3 Current Research . . . 3
- 1.4 Overview of Thesis . . . 4

2 Constructive Proof Systems **6**
- 2.1 Natural Deduction Proofs . . . 6
- 2.2 Programming Using Constructive Proofs . . . 7
- 2.3 Correctness . . . 10

3 Network Model **12**
- 3.1 Network Model Definition . . . 12
- 3.2 Network Primitives . . . 13
  - 3.2.1 Specification of Adding a Station to a Network . . . 17
  - 3.2.2 Specification of Removing a Station from a Network . . . 19
  - 3.2.3 Specification of Connecting Stations in a Network . . . 21
  - 3.2.4 Specification of Disconnecting Stations in a Network . . . 23
  - 3.2.5 Additional Axioms and Theorems . . . 25

4 Operation Modelling Method **29**
- 4.1 Method Description . . . 29

5 Complex Network Operations **31**
- 5.1 Forced Connection . . . 31
  - 5.1.1 Motivation . . . 31
  - 5.1.2 Defining a Forced Connection . . . 31
  - 5.1.3 Proving the Forced Call Operation . . . 33
  - 5.1.4 Problems Encountered During Proof . . . 34
- 5.2 Call Attempt . . . 35
  - 5.2.1 Motivation . . . 35
  - 5.2.2 Defining a Call Attempt . . . 35
  - 5.2.3 Proving the Call Attempt Operation . . . 37
  - 5.2.4 Problems Encountered During Proof . . . 37

6 Individual Features **38**
6.1 Call Display 38
6.1.1 Defining Call Display 38
6.1.2 Proving the Call Display Feature 38
6.2 Call Block 40
6.2.1 Defining Call Block 40
6.2.2 Proving the Call Block Feature 41
6.3 Feature Interaction Prevention Method 43

7 Conclusions **45**
7.1 Feature Interaction Specifications in MIZAR-C 45
7.2 Contributions to Research 46
7.3 Future Work 46

A Network and Station Extensions **48**

B Primitive Operations **54**
B.1 Specification of Primitive Operations 54
B.2 Proof of NET_Add_Station 55
B.3 Proof of NET_Remove_Station 57
B.4 Proof of NET_Idle_Connect 59
B.5 Proof of NET_Busy_Connect 61
B.6 Proof of NET_Disconnect 63

C Proof of Forced Connection **66**
C.1 Pseudo-Proof of Forced Connection 66

D Proof of Basic Call Attempt **70**
D.1 Pseudo-Proof of Call Attempt 70

E Feature Formalization **74**
E.1 Proof of Call Display 74
E.2 Proof of Call Block 76

F Executions of Extracted Code **79**
F.1 Forced Connection 79
F.2 Basic Call Attempt 81
F.3 Call Display 82
F.4 Call Block 84

Bibliography **85**

List of Figures

4.1 Operation Modelling Method 29

5.1 Forced Connection Theorem 32
5.2 Consistency Specification 34
5.3 Call Attempt Theorem 36
5.4 Call Attempt Redefinition 37

6.1 Call Display Theorem 39
6.2 Network Model Changes for Call Display 40
6.3 Call Block Theorem 41
6.4 Network Model Changes for Call Block 42
6.5 Network Model Changes for Unique Identifiers 43
6.6 Feature Interaction Prevention Method 44

7.1 Feature Interaction Prevention Method 45

List of Tables

2.1 Proof Construct Realizations 8

Chapter 1

Introduction

1.1 Motivation

In the telecommunications world, a feature is functionality that can be incrementally added to enhance or modify a network service. A feature may interact with the telecommunications system, other features and services, and other instances of the same feature in a way that interferes with the feature's desired method of operation (see Bowen et al. [4]). Every new feature added can cause changes in the behavior of other features or even cause them to fail in such a way that the system crashes. With multiple telecommunications vendors and configurations, adding new features becomes more frequent and complex. The *feature-interaction problem* is the detection, prevention, and resolution of undesired feature interactions.

Velthuijsen [20] classifies examples of the feature-interaction problem into three areas: *conflict*, *ambiguity*, and *resource contention*. *Conflict* is the problem of two features containing responsibilities which conflict with each other. *Ambiguity* occurs when two or more features use a single signal for determining their course of action. When the signal occurs and multiple features are involved, the outcome may be unpredictable. *Resource contention* occurs when multiple features require some resource from the telecommunications system and the demand is greater than the supply of the resource.

For example, a *conflict* may occur when a subscriber to the telecommunications system has an unlisted number. This subscriber makes a call to another subscriber who has the caller identification feature. A conflict may occur here because one feature requires a directory number to be unlisted (thus it should not be identified) while another feature at the same time requires that the number be identified. A method of resolution would be to provide an alternative piece of information to identify the caller with an unlisted number. An example would be to list the caller as "unknown".

Ambiguity may occur when a subscriber has both the call-waiting and multiway-calling features. When the subscriber of call-waiting is currently involved in a call and a second call is made to the subscriber, a warning tone is made and the subscriber is able to use the flashhook signal to switch between the callers. On the other hand,

a flashhook signal tells the multiway-calling feature that a third party should be included in the current call (conference call). When these two features are active and a flashhook signal occurs there is ambiguity concerning the desired course of action.

Another feature interaction may occur using the call-waiting and multiway-calling features. Both of these features require a bridge (a device for merging calls) as a resource. This resource is limited in telecommunications systems. Thus, there is *resource contention* between these two features and the limited resource (the bridge). A method of resolution would be to have one feature disable the other feature on it's activation (i.e. only one feature active at a time).

These feature interactions are a key problem in the telecommunications industry. The current trend is to rapidly add new features to the public telephone networks to meet user needs. Since so many features are added so quickly, very little time is spent on ensuring that the features do not interact. This can lead to feature failures during use.

Griffeth and Lin [11] have identified how these feature-interaction problems can be formulated to be of interest to computing scientists. Their interpretations are familiar:

- **Software Maintenance Problem.** The problem of modifying existing software systems in an efficient manner.
- **Legacy System Problem.** How do we adapt existing systems to the ever changing technologies and markets?
- **Software Reuse Problem.** The ability to support reuse of existing software systems in different systems.

These are well-known difficult problems. By focusing our attention on the feature-interaction problem, we hope to provide some insight into these larger problems.

1.2 Formalizing Telephony Features

As discussed, there are three major classifications of feature-interaction problems: *conflict, ambiguity, and resource contention.* The key to predicting these behaviors is in feature specification. Since feature specification causes these interactions, the undesirable ones should be detected at specification time.

Essentially there are two ways to deal with feature interactions. The first is to specify the system and reason about it using some brand of logic and semantics. This way, the system is designed so that interactions cannot occur due to architectural reasons. The second approach is to detect design errors leading to feature interactions. This can be done by specifying the system and using simulations to weakly verify the design.

The research presented in this thesis is based on a variation of the first approach where an attempt is made to model the features so that unwanted interactions do

not occur. What makes our approach unusual is the use of constructive methods to derive operations within the system. Not only do we specify and reason about the features, but we actually generate the code that implements them. We are able to create formally specified features and corresponding interactions which do not cause unwanted results.

One naive approach to the feature interaction problem would be to completely specify all features and interaction resolutions prior to implementation. Some of the causes of feature interactions may be difficult to deal with at the feature design phase (see Cameron et al. [6]). The difficulty is that different customers may have different preferences for the same features. This indicates that a single policy governing the availability of features is unrealistic and the existence of a set of rigid precedence relations is infeasible for resolving conflicting call control strategies. It may also be the case that instantiations of feature parameters may be so unpredictable that dealing with all possible cases before the deployment of features is impossible. But the real problem is that features are developed incrementally over the life of a system and so any method must take this practical consideration into account.

1.3 Current Research

Current formal approaches to feature interactions focus on the detection and resolution of feature interactions. Features are specified in some formal method language and interactions are detected by the system. Once a feature interaction has been detected, some method of re-specification is used to provide a resolution.

The feature interaction literature includes a number of approaches that address the feature interaction problem. The following describes four of the current approaches.

Language Of Temporal Ordering Specifications (LOTOS). LOTOS is a specification language developed by the International Organization for Standardization (IOS) for the specification of open systems interconnection (OSI) services and protocols. It has a formal basis and has been used for different types of distributed systems. If an appropriate specification style is used, LOTOS specifications can be executable. This approach is based on using algebraic data types for reasoning.

Boumezbeur and Logrippo [3] and Faci and Logrippo [9] use LOTOS to specify a telephone system involving sets of features. LOTOS tools are used to develop executable specifications. These executables are then simulated to detect design errors leading to feature interactions. Unfortunately, it is not guaranteed to capture all errors.

Input/Output Automaton.

Fekete [10] (as well as Zave [21]) takes the approach of applying Finite State Automata (FSA), in particular I/O Automata, to communication services. The I/O Automaton model describes a specification by giving a state-transition machine that generates the required sequence of actions.

SDL Specification Language.

The Specification and Description Language (SDL) is based on the Extended Finite State Machine. SDL is a graphical, formal, object-oriented specification language for real-time systems. A specification in SDL is composed of parallel processes communicating by FIFO queues. SDL uses an abstract data type approach to data specification. Abstract data types are creating using data types, operators of those types, and axioms about the operators.

Combes and Pickin [8] present a method for detecting feature interactions by verifying that service feature requirements, specified in a temporal logic, are satisfied on a formal specification language (SDL) model of the network and the service features concerned.

Behavioral Chaining.

Cheng [7] presents a technique for incremental service specification and interaction management. He identifies behavior chaining where a constraint-oriented style of specifications is used to manage service interaction. His approach is specified in LOTOS (see above).

Each new service is specified independently based on a common model and/or existing services and features. He achieves this by chaining together existing and new service/feature behaviors to create the required new service. Constraints may be used during the composition to manage and resolve service interactions.

1.4 Overview of Thesis

My research takes on a different approach than what is being studied by current researchers. Current research tends to focus on the detection and resolution of the feature interactions. My research attempts to specify the features such that interactions that do occur shall occur in a predictable way.

A formal logical language called MIZAR-C is used to specify different interacting features. By creating a model of these features and interactions in MIZAR-C, I have been able to develop a method used to identify feature interactions and provide a resolution in the specification. These formal specifications can then be used to produce executable interacting features with a guarantee that results of interactions are predictable.

Although there are a number of different types of interactions that can occur in a telephony network, my research focuses on the conflicting interactions. This occurs when two features have opposing definitions. Although my research is restricted to these type of interactions, the methodology presented here should hold for all types of feature interactions.

The rest of this thesis outlines the details of the project and the conclusions reached.

Chapter 2 identifies background information necessary to understand the foundations of constructive systems such as MIZAR-C.

Chapter 3 describes the development of the telephony network model being researched. The basic network primitives are outlined in Appendix A and B.

Chapter 4 discusses the method used to produce specifications of operations in MIZAR-C.

Chapter 5 details two complex operations. These proofs are outlined in Appendix C and D.

Chapter 6 details the approaches taken to develop proofs of individual features. The proofs of these individual features are laid out in Appendix E.

Chapter 7 discusses the conclusions made about this approach and future work considerations are explored.

Chapter 2

Constructive Proof Systems

2.1 Natural Deduction Proofs

MIZAR-C allows natural deduction proofs to be completed where code can be extracted from the constructive parts of the proof. Natural deduction is based upon axioms and a collection of inference rules. Axioms are built by making assumptions and using inference rules to form conclusions. If we are reasoning about formulas, we can form implications. As an example:

Implication Proof

let x
assume $Pre[x]$
.
.
.
consider y
.
.
.
thus $Post[x, y]$

$\forall x Pre[x] \Rightarrow Post[x, y]$

This proof introduces a universal variable, x, and then assumes the preconditions, $Pre[x]$. In the reasoning of the proof, an existential quantifier is introduced to identify variable, y. Finally, the postconditions, $Post[x, y]$ are produced. This reasoning would then prove the introduction of an implication, $\forall x Pre[x] \Rightarrow Post[x, y]$.

Realizations (or code extraction) is based on the identification of the constructive content of arithmetical statements. This was first introduced by Kleene [15]. Every logical formula has an equivalent computation (or realization). Only formulas that have inputs contain computational content and this is in the form of realizations. Formulas that are simply constants or expressions that require evaluation to produce a

constant, lack computational content. For other work exploring methods of optimizing extracted programs see [16] [1] [19].

In this thesis, the following conventions are used when presenting examples of statements and proofs done in MIZAR-C:

- The type style *for x being Tx holds P[x]* will be used when writing examples of MIZAR-C proof text.
- The type style `(LAMBDA () x)` will be used in the examples of realizations of MIZAR-C proof text.

2.2 Programming Using Constructive Proofs

Before using proofs to create programs and even before we can reason about software systems in logic, it is necessary to construct specifications of the software. A specification of a software module identifies a set of pre-conditions and a set of post-conditions. A specification can be expressed as follows:

$$\forall x(Pre[x] \Rightarrow \exists y Post[x, y])$$

This expression says that if a set of pre-conditions holds then the post-conditions must follow. The pre-conditions identify a set of states that must be true prior to applying the theorem. The post-conditions identify changes that occur to the states if the pre-conditions held. This is a basic definition of any software module. This can be taken one step further to describe an actual function (or program), say f, so that

$$\forall x(Pre[x] \Rightarrow Post[x, f(x)])$$

This statement only gives the ability to reason about the specification of a software module. Therefore, all that is interesting is how the function performs or utilizes resources. In order to use proofs as a method of programming, a programming task exists in which it becomes necessary to prove:

$$\exists f \forall x(Pre[x] \Rightarrow Post[x, f(x)])$$

In order to accomplish this, proofs need to be interpreted as software modules. A relationship is necessary between proof constructs and programming constructs. This is called *realizability interpretation* and the code generated from the logical construct is called its *realization*. The following identifies the common proof constructs and the corresponding realization:

Proof Construct	Program Construct
$\forall$ introduction	Function Abstraction
$\forall$ elimination	Function Application
$\exists$ introduction	Computation Definition
$\exists$ elimination	Evaluation
$\Rightarrow$	Type Guarding
induction	Recursion

Table 2.1: Proof Construct Realizations

To further the understanding of these realizations, the following are some examples of the general analogies that are identified in Table 2.1.

Proofs	Programs
$\forall$ intro: let x be int; . . . thus $P[x]$ / $\forall x P[x]$	**function declaration:** `f(int x){` . . . }
$\forall$ elim: $\forall x P[x]$ / $P[6]$	**function application:** `f(6)`

Universal introduction can be viewed as introducing a function, and eliminating the universal variable is analogous to applying the function to a specific argument.

$\exists$ intro:

$$\frac{P[6+g(2)]}{\exists x\ P[x]}$$

results:

```
f(){
  .
  .
  .
  return 6+g(2);
}
```

$\exists$ elim:

$$\frac{\exists x\ P[x]}{\text{consider } t \text{ st } P[t]}$$

assignment:

```
t = f();
```

Existential introduction can be viewed as creating a macro that uses global arguments to compute a specific value, and the existential elimination can be viewed as assigning the value computed by the macro to a particular variable.

$\Rightarrow$ intro:

assume M
.
.
.
thus Q

$$\frac{}{M \Rightarrow Q}$$

use a module:

```
#include M
.
.
.
module Q
```

$\Rightarrow$ elim:

$$\frac{M,\ M \Rightarrow Q}{Q}$$

linking/loading:

```
ld Q.o -o Q -lM
```

One way to view implication introduction is as follows: given a set of facts M (a module), one can use them to produce another module, Q. Implication elimination can be then seen as using an existing module to create another module.

$\vee$ intro:

$$\frac{P \Rightarrow R \qquad Q \Rightarrow R}{P \vee Q \Rightarrow R}$$

case analysis:

```
switch(?)  {
  case P: R
  case Q: R
}
```

$\vee$ elim:

$$\frac{P \vee Q,\ P \Rightarrow R,\ Q \Rightarrow R}{R}$$

case execution:

```
execute above code
```

These two rules relate to making decisions in a procedure. The first creates the code to perform the decision, while the second would actually execute the procedure to produce the desired result.

induction:

let x
assume Ihyp:
$\forall y \,.\, y < x \Rightarrow P[y]$
.
.
.
thus $P[x]$

$$\frac{}{\forall z\ P[z]}$$

recursion:

```
int f(int x) {
        .
   ...  f(x-1) ...
        .
   ...  f(x-3) ...
        .
  return value of f(x)
}
```

An inductive proof results in a recursive function that can use the values from the recursive calls to compute the desired output. The structure of the proof would determine the base case and the recursive calls to be made.

2.3 Correctness

The whole intent of using formal methods is to specify and verify software systems. The very first question that should arise is what is the amount of correctness provided by using a formal system. This question is the same as asking how convinced we are of the truth of:

$$\exists f \forall x (Pre[x] \Rightarrow Post[x, f(x)])$$

The answer to this lies in the approach taken to model a real world application in a particular formal system.

There are two approaches to using formal methods to model applications. The first approach (explicit) develops the function and the correctness proof simultaneously. This requires that general programming be done since some programming constructs are difficult to reason about. At the same time, the corresponding realization may not be implemented correctly and may cause errors. The second approach (implicit) is to reason about a problem that has been modelled in a formal logic system and then extract the program automatically from the structure of the proof. The implicit approach is used in the MIZAR-C system.

The main advantage of using formal logic is that once a theorem is proven, it is correct. The one weak area in using formal logic, particularly in the implicit approach, is that a model must be developed which represents a real world problem. Since it is the specification of a system that defines the model, it is up to the user doing the

specifications to model the real world as correctly as possible. Incorrect modelling can lead to incorrect specifications. We may still be able to prove a specification but in essence we have proven the wrong thing since the model is incorrect.

There are two ways to model real world applications in the implicit approach. The first is to base the entire formal system on a small set of primitives (e.g. Peano Axioms). Using this approach requires that real world applications be modelled based on the primitive axiom set. This may be a disadvantage since abstract data types, which represent real world entities, have to be derived from the primitive foundation. The second approach is when extensions are allowed to be created in the formal system. This approach allows the creation and reasoning of the abstract data types by introducing a set of axioms and implemented primitive operations which are native to the domain of the problem.

The realizability interpretations alone do not produce computation in MIZAR-C. Extensions must be added in order to produce formulas with content. Without these extensions, the programs extracted from MIZAR-C are simply functions with no objects to apply them to. Of course, these extensions must then be hand-verified for consistency and completeness.

In my opinion, correctness can be defined in the following three levels:

Low levels of correctness exist in systems that can not verify all possible conditions (feature interactions). LOTOS, which uses simulations in an attempt to detect and resolve feature interactions, does not guarantee to capture all feature interactions. VDM and Z are formal languages which do not provide an automated checker and allow errors in specifications to be introduced by the user [12].

Moderate levels of correctness exist in systems that provide some method of automation but not necessarily a complete and correct specification. MIZAR-C is a system which provides the automation of proof checking but since primitive implementations and the existence of extensions are allowed, the amount of correctness is still limited. This is due to the need to hand-verify parts of the specification.

High levels of correctness exist in systems that provide automated proof verification and base the entire formal system on proven primitives. PC-Mizar is a tool which bases all reasoning on a mathematical foundation and also provides a proof checker. This mathematical foundation is a groundwork for building bigger and better theorems. Since there is no introduction of hand-verified axioms or theorems, these types of systems provide the highest degree of correctness.

Although these definitions may generalize correctness levels too much, they do provide a guideline for determining the amount of correctness a system provides. It must also be kept in mind that as the level of correctness increases, the complexity of completing a project increases. For these reasons, MIZAR-C is probably the best choice for this feature interaction project. It provides a moderate amount of correctness while at the same time provides a reasonable amount of complexity.

Chapter 3

Network Model

3.1 Network Model Definition

The first step in reasoning about a telephone network is to model the network. This requires evaluating the telephone system and extracting information from real telephone networks. A network is made up of a number of stations (or telephones) which have characteristics defined by connections with other stations and the features that they may have enabled. It is these networks, stations, connections, and features that need to be defined in a model.

For our purposes, a network is simply a set of stations. A network may be empty or contain numerous stations. Stations in a network are unique (i.e. one station cannot be represented twice in a network). Operations that can be performed on a network are to add/remove stations and to establish point-to-point connections.

A station is a representation of a telephone from a real telephony network. Each station has state characteristics. For example, one state characteristic identifies if the station is busy while other characteristics may declare that specific telephony features are turned on. A station is considered busy if it is involved in a connection with another station. For simplicity, any one station can only be involved in one connection. Stations are considered to be unique and also maintain an identity. Operations that can be performed on stations include connecting and disconnecting two stations.

A connection occurs when two stations are connected which allows the two stations to communicate to each other. Since stations can only be involved in one connection at any given time, another station should not be allowed to physically connect to a busy station. The only time this changes is if features such as call conferencing are introduced, in which case, a special type of station may have to be introduced which allows all stations involved in a conference to connect to it. This would still enforce the rule that all "real" stations maintain the characteristic that they can only be connected to one other station at any given time.

Features are modelled as side effects of calling operations. Features in a telephony network are based upon the existence of a connection between two stations. Since

one station can call another station, certain conditions or activities may occur due to the state of a station's set of features. Examples of features include call display, call block, call conferencing, etc.

An important concept that needs to be modelled is the ability for operations on the network to maintain consistency. One approach would be to identify a single network that is modified by operations in time. The operations would be specified in such a way that modifications never destroy the consistency of the network. Unfortunately, there does not exist a method to model time in MIZAR-C logic.

The approach taken is one of using network (state) transitions. In this way, operations are given a network in which modifications can take place and a new network is returned. The specifications of the operations ensure that the new network being returned is consistent. This approach avoids the use of time.

3.2 Network Primitives

In order to be able to produce realizations of proofs pertaining to the feature interaction problem, it was necessary to develop an extension to the MIZAR-C system. The extensions identify the basic data types in the telephony network being modelled and the implemented primitive operations which can be performed on it. The basic data types are *Networks* and *Stations*. The axioms and implementations of the basic primitives constitute the minimal set of primitives. From the minimal set, the features and feature interactions could be derived.

The reason for a minimal set is that the specification represents the model of the real system. If it is a large model, it becomes too complex and may increase the possibility of an incorrect specification. At the same time, if the specification is large, it may be difficult to ensure that the implemented primitive operations are consistent with the specifications.

The basic data types are the *Network* and *Station*. A Network represents a set of stations and all the properties of the stations and connections in the network. In MIZAR-C the *given* statement is used to introduce data types, predicates, and operations on those data types.

```
/* Introduce the Station data type. */
given Station being [Any ];

/* Introduce the Network data type. */
given Network being [Any ];
```

From these data types, predicates were defined for *Busy* and *IsConnected*. These predicates indirectly access the properties of the network model. Busy identifies that a station is busy in the network (or involved in a connection). IsConnected indicates that two stations in a given network are connected to each other.

```
/* Busy [N, S] is true if station S is busy in network N.                    */
given Busy being [Network, Station ];

/* IsConnected[N, S1, S2] is true if stations S1 and S2 are connected in */
/* network N.                                                              */
given IsConnected being [Network, Station, Station ];
```

These two predicates are used in order to determine if a given station is busy or not and also if two stations are connected or not. Since the predicates only identify the characteristic of being busy (or connected) it does not determine whether a particular station(s) is busy (connected). In other words, in order to reason about the state of the network, we need to be able to access the state. This requires the introduction of *decider axioms*. Since these are deciders, there is constructive content. Therefore, these deciders are realized by implemented primitive operations.

```
/* For every network N and station A, A is either busy in N or it is not.*/
/* This statement is realized by an implemented operation called          */
/* networks-Busy_or_not.                                                  */
NET_Busy_or_not: (for N being Network holds
  (for A being Station holds
    ( Busy[N, A ] or (not Busy[N, A ]) ) ) )
rby networks-Busy_or_not;
{<<networks-Busy_or_not:networks-Busy_or_not>>}

/* For every network N and stations A and B, A and B are either connected*/
/* in N or they are not.                                                  */
/* This statement is realized by an implemented operation called          */
/* networks-IsConnected_or_not.                                           */
NET_IsConnected_or_not: (for N being Network holds
  (for A, B being Station holds
    ( IsConnected[N, A, B ] or (not IsConnected[N, A, B ]) ) ) )
rby networks-IsConnected_or_not;
{<<networks-IsConnected_or_not:networks-IsConnected_or_not>>}
```

MIZAR-C uses the *rby* syntax to show that a statement is realized by a particular implementation. In this case *NET_Busy_or_not* is realized by the implemented primitive *networks-Busy_or_not*. *NET_IsConnected_or_not* is realized by the operation *networks-IsConnected_or_not*. In MIZAR-C operations generally do not have content since the operation is the actual content. Deciders (disjunctions) or statements which make decisions (existentials) normally have content since this is the interface between run time and the "program" in logic.

At this point, two other predicates were introduced, *ContainedIn* and *IsEmpty*. ContainedIn defines when a station is contained in a network. When a network does not contain any stations IsEmpty is considered to be true.

```
/* ContainedIn[N, S] is true if station S is contained in network N.      */
given ContainedIn being [Network, Station ];

/* IsEmpty[N] is true if N is a network with no stations.                 */
given IsEmpty being [Network ];
```

The Busy and IsConnected predicates are used as deciders to determine whether or not the predicate is true. On the other hand, these new predicates, *ContainedIn* and *IsEmpty*, were introduced to be used in a different way. In the last case, they were used as deciders in order to determine the state of network. In this case, they are not used as deciders but rather guards. A guard is an antecedent of an implication statement. Since they are only guards, there is no computation content.

Another condition that comes from the problem definition is that stations which are connected are busy. This requires that we add another axiom, *NET_IsConnected_Busy*, to the specification. It indicates that if two stations are connected then they are both busy.

```
/* For every network N and every two stations A and B, if A and B are     */
/* connected in N then A is busy in N and B is busy in N.                 */
/* This statement contains no constructive content.                       */
NET_IsConnected_Busy: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies ( Busy[N, A ] & Busy[N, B ] )) ) );
```

So far from this specification, we have defined the data types that we are using, basic state characteristics and some method of accessing the states. The next step is to evaluate what type of primitive operations may be performed on the network and stations. In a telephony network it is necessary to be able to *add*, *remove*, *connect*, and *disconnect* stations. These are the primitive operations of a network and in order to reason about operations in the modelled network, these operations must be introduced.

The following MIZAR-C definitions introduce the four primitive operations. Since they must be implemented, the *rby* syntax is used to bind the proof names to code names. The *add_station* statement is bound to *networks-add_station* code and so on.

```
/* The primitive operation add_station<N, A> returns a new network        */
/* which represents the network N with the exception that the station A   */
/* has been added.                                                        */
/* This operation is realized by the implemented operation called         */
/* networks-add_station.  The primitive operation must be hand-verified   */
/* to ensure it matches its specifications.                               */
given add_station being (<Network, Station > -> Network)
  rby networks-add_station;
{<<add_station:networks-add_station>> }
```

```
/* The primitive operation remove_station<N, A> returns a new network    */
/* which represents the network N with the exception that the station A  */
/* has been removed.                                                     */
/* This operation is realized by the implemented operation called        */
/* networks-remove_station.  The primitive operation must be hand-        */
/* verified to ensure it matches its specifications.                     */
given remove_station being (<Network, Station > -> Network)
  rby networks-remove_station;
{<<remove_station:networks-remove_station>> }

/* The primitive operation connect<N, A, B> returns a new network        */
/* which represents the network N with the exception that the stations A */
/* and B are connected.                                                  */
/* This operation is realized by the implemented operation called        */
/* networks-connect.  The primitive operation must be hand-verified to   */
/* ensure it matches its specifications.                                 */
given connect being (<Network, Station, Station > -> Network)
  rby networks-connect;
{<<connect:networks-connect>> }

/* The primitive operation disconnect<N, A, B> returns a new network     */
/* which represents the network N with the exception that the stations A */
/* and B are disconnected.                                               */
/* This operation is realized by the implemented operation called        */
/* networks-disconnect.  The primitive operation must be hand-verified   */
/* to ensure it matches its specifications.                              */
given disconnect being (<Network, Station, Station > -> Network)
  rby networks-disconnect;
{<<disconnect:networks-disconnect>> }
```

These primitive operations return a new network which is based on the old with the exception(s) of the changes made by the operation. The decision to base operations on network state transitions was made due the fact that there is no method in MIZAR-C logic to model changes taking place in an object over time.

The operations are only introduced in the above statements. In order to reason about them and also to implement the operations, axioms need to be introduced which specify the behavior of the operations. When axioms are identified to specify the operations, the user must ensure that the implementations match the corresponding axioms. If this is not done, the final implementations derived from any proofs using these operation may be incorrect.

The following subsections describe the specifications of the primitive operations as well as some other axioms and theorems which were added. As features were developed, other axioms were required to extend the base model but these will be discussed in the section describing the features. A complete listing of the axioms is

contained in Appendix A.

This axiom set is the basis of the telephony network model. Upon this model computations can be built involving *networks*, *stations*, and respective connections. This leads to the development of the features described in Chapter 6.

3.2.1 Specification of Adding a Station to a Network

For all the operations, the first axiom introduced is one which identifies the use of the operation. For the *add_station* operation we introduce *NET_Add_station_identity* which states that the operation may be applied to any network with any station that is not already contained in the network. The guard statement that the station does not already exist in the network is unnecessary since a station that already exists could, in principle, be added without any problems. The decision to add this was made to help clarify the operation. The statement *((add_station <N, A >) = (add_station <N, A >))* is the method in which MIZAR-C indicates the definition of the operation over the domain of all network-station pairs.

```
/* For every station A that is not contained in network N, there is an   */
/* operation, add_station<N, A>, which adds A to N and returns a new     */
/* network.                                                              */
/* This statement contains no constructive content.                      */
NET_Add_station_identity: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A ]) implies
      ((add_station <N, A >) = (add_station <N, A >))) ) );
```

The *add_station* operation must be specified to return a new network in which the new station has been added.

```
/* For every station A not contained in network N, A will be contained   */
/* in the new network returned from the add_station<N, A> operation.     */
/* This statement contains no constructive content.                      */
NET_Add_station_contained: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A ]) implies
      ContainedIn[(add_station <N, A >), A ]) ) );
```

These previous two axioms are really the definition of how the operation works. One concern is that those axioms by themselves do not preserve the state of the rest of the network. This issue of maintaining consistency can be addressed in a couple of ways. The first approach is to define an axiom describing a consistent network. From the network model definition the basic characteristics are that any one station can only be involved in one connection at any given time and that no station can connect to itself. This could be stated as follows:

Consistent_Network: (for N being Network holds
(for A, B, C being Station holds
(IsConnected[N,A,B] & IsConnected[N,C,B]) implies
(A = C)) &
(for A, B being Station holds
(IsConnected[N,A,B]) implies
(A <> B)));

I have explored this method of modelling the network in another system, PC-Mizar [2] [17] [18] (PC-Mizar will not be detailed in this thesis). It has been found that as new features are added to the network, the definition of a consistent network must reflect the characteristics of the new features. In other words, if an operation is being performed on a given network, the feature states (ON or OFF) of stations in the original network must remain the same in the resulting network unless those stations were involved in the operation.

This has been shown to be a very tedious approach to modelling the network. Whenever a new feature is added, a new corresponding consistency characteristic must be added to the definition of consistency. Furthermore, all proofs of previous features/operations must reflect this change to ensure that an operation does not contradict the definition of a consistent network. Although it is tedious, it ensures a higher level of correctness.

In the second approach, an assumption is made that all feature characteristics are maintained throughout operations unless they are directly involved in the operation. Taking this approach means that all specifications must be validated by hand to ensure that obscure operations are never performed (i.e side effects). The advantage to this is that a simple definition of a consistent network can be specified from the beginning and maintained throughout the project. Although this is not the most correct approach, it provides a method with a low level of correctness to quickly model and reason about a telephony network. This is the approach taken in this project.

Two axioms were introduced to ensure that the connections and membership of the old network were maintained when the *add_station* operation was performed. The first axiom states that after adding a station, all stations except the one that was added, that were connected in the old network are still connected in the new network. The second axiom ensures that the stations from the old network are still in the new network and that the added station is also in the new network. The second axiom is the only one which requires an implementation since it makes a decision. The decision made is to determine if a station in the new network is one from the old network or the new station that was added.

```
/* For every station A not contained in network N, the new network      */
/* returned from the add_station<N, A> operation will maintain all the  */
/* connections that existed in N.                                       */
/* This statement contains no constructive content.                     */
```

```
NET_Add_station_connections: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A ]) implies
      (for X, Y being Station holds
        (( (X <> A) & (Y <> A) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[(add_station <N, A >), X, Y ])) )) ) );

/* For every station A not contained in network N, the new network      */
/* returned from the add_station<N, A> operation will contain all the   */
/* stations from N and also A.                                          */
/* This operation is realized by the implemented operation called       */
/* networks-Add_station_remains_same.                                   */
NET_Add_station_remains_same: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A ]) implies
      (for Z being Station holds
        (( ContainedIn[N, Z ] or (Z = A) ) iff
        ContainedIn[(add_station <N, A >), Z ]) )) ) )
rby networks-Add_station_remains_same;
{<<networks-Add_station_remains_same:networks-Add_station_remains_same>>}
```

By defining consistency in these primitive operations and making the assumption that feature states are maintained, it is possible to build other operations and features of the network without the need to redefine consistency. From these basic primitives other operations and features were built.

3.2.2 Specification of Removing a Station from a Network

As with the *add_station* operation, the first characteristic to be stated is one which defines the existence of the operation. In this case the axiom, *NET_Remove_station_identity*, states that the operation may be applied to any network and station where the station is in the network. The guard statement that the station must be contained in the old network is here out of convenience. The other part of the guard statement states that the station to be removed cannot be busy. This avoids the problem of specifying what happens to connections with the removed station.

```
/* For every station A that is contained in network N, there is an      */
/* operation, remove_station<N, A>, which removes A from N and returns a */
/* new network.                                                         */
/* This statement contains no constructive content.                     */
NET_Remove_station_identity: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N,A] & (not Busy[N, A])) implies
      ((remove_station<N,A>) = (remove_station<N,A>)))));
```

The *remove_station* operation must be specified to return a new network in which the station has been removed.

```
/* For every station A contained in network N, A will not be contained in*/
/* the new network returned from the remove_station<N, A> operation.     */
/* This statement contains no constructive content.                      */
NET_Remove_station_not_contained: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N, A ]& (not Busy[N, A])) implies
      (not ContainedIn[(remove_station <N, A >), A ])) ) );
```

Two axioms were introduced to ensure that the connections and membership of the old network were maintained when the *remove_station* operation was performed. The first axiom states that after removing a station, all connections except any involving the removed station that were connected in the old network are still connected in the new network. The second axiom ensures that the stations from the old network are still in the new network and that the removed station is not in the new network. The second axiom is the only one which requires an implementation since it makes a decision. The decision made is to determine if a station in the old network is one that is in the new network or that it is the old station that was removed.

```
/* For every station A contained in network N, the new network returned  */
/* from the remove_station<N, A> operation will maintain all the         */
/* connections that existed in N with the exception of any involving A.  */
/* This statement contains no constructive content.                      */
NET_Remove_station_connections: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N, A ] & (not Busy[N, A])) implies
      (for X, Y being Station holds
        (( (X <> A) & (Y <> A) ) implies
          (IsConnected[N, X, Y ] iff
          IsConnected[(remove_station <N, A >), X, Y ])) )) ) );

/* For every station A contained in network N, the new network returned  */
/* from the remove_station<N, A> operation will contain all the stations */
/* from N except A.                                                      */
/* This operation is realized by the implemented operation called        */
/* networks-Remove_station_remains_same.                                 */
NET_Remove_station_remains_same: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N, A ] & (not Busy[N, A])) implies
      (for Z being Station holds
        (ContainedIn[N, Z ] iff
        ( ContainedIn[(remove_station <N, A >), Z ] or (Z = A) )) )) ) )
rby networks-Remove_station_remains_same;
```

```
{<<networks-Remove_station_remains_same:networks-Remove_station_remains_same>>}
```

3.2.3 Specification of Connecting Stations in a Network

In specifying the operation of connecting two stations, two scenarios had to be addressed. The first is when both stations are idle. In which case, the connection can be completed. In the case that either or both of the stations are busy, the connection cannot be completed and the network should be maintained.

As with all the primitive operations, the first axiom for connecting station is to identify the existence of the operation. In this case, *NET_Idle_connect_identity* states that if the two stations are not busy then the operation can be performed to connect them. In this case, the guard statement is necessary to indicate that the stations were not busy in the previous network. Unlike the previous identity axioms, this guard cannot be removed.

```
/* For every two stations A and B that are not busy in network N, there  */
/* is an operation, connect<N, A, B>, which can be used to connect A and */
/* B and return a new network.                                           */
/* This statement contains no constructive content.                      */
NET_Idle_connect_identity: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
      ((connect<N,A,B>) = (connect<N,A,B>))))));
```

The *connect* operation must be specified to return a new network in which the stations have been connected if they were idle in the previous network.

```
/* For every two stations A and B that are not busy in network N, A and  */
/* B are connected in the new network returned from the connect<N, A, B> */
/* operation.                                                            */
/* This statement contains no constructive content.                      */
NET_Idle_connect_is_connected: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
      IsConnected[(connect <N, A, B >), A, B ]) ) );
```

The next axiom added ensures that connections are not disrupted from the previous network.

```
/* For every two station A and B that are not busy in network N, the new */
/* network returned from the connect<N, A, B> operation, will maintain   */
/* all the connections that existed in N.                                */
/* This statement contains no constructive content.                      */
```

```
NET_Idle_connect_connections: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
      (for X, Y being Station holds
        (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ])) )) ) );
```

The following axiom states that busy stations in the previous network are still busy in the new network. This is actually an unnecessary axiom since we have stated previously that if stations are connected then they are also busy. The reason this was added was that it saves some steps in proofs. If it was missing and we wanted to prove that a station from the previous network was busy, we would first have to prove that there exists another station which the first one is connected to. Then, we would be able to use the previous axiom to show that since it is connected, it is also busy. With this new axiom, we can avoid that problem completely.

```
/* For every two stations A and B that are not busy in network N, the    */
/* network returned from the connect<N, A, B> operation will maintain    */
/* the busy state of all stations in N.                                  */
/* This statement contains no constructive content.                      */
NET_Idle_connect_remains_busy: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
      (for Z being Station holds
        (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ])) )) ) );
```

The following axioms define the characteristics of the *connect* operation when one or both of the stations are busy. It states that if either of the stations are busy then the *connect* operation just returns the original network. The guard statement indicating that one or both of the stations is busy is necessary.

```
/* For every two stations A and B where either A is busy in network N or */
/* B is busy in N, there is an operation, connect<N, A, B>, which can be */
/* used to attempt to connect A and B and return a new network.          */
/* This statement contains no constructive content.                      */
NET_Busy_connect_identity: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies (connect<N,A,B>) = N)));
```

The following axioms show that the new network remains the same as the previous network. So, connections are maintained. Once again, an axiom was added showing that busy stations remain busy to simplify proofs.

```
/* For every two stations A and B where either A is busy in network N or */
/* B is busy in N, the new network returned from the operation           */
/* connect<N, A, B> maintains all the connections that existed in N.     */
/* This statement contains no constructive content.                      */
NET_Busy_connect_connections: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies
      (for X, Y being Station holds
        (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ]) )) ) );

/* For every two stations A and B where either A is busy in network N or */
/* B is busy in N, the new network returned from the connect<N, A, B>    */
/* operation maintains the busy state of all stations in N.              */
/* This statement contains no constructive content.                      */
NET_Busy_connect_remains_busy: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies
      (for Z being Station holds
        (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ]) )) ) );
```

3.2.4 Specification of Disconnecting Stations in a Network

Once again, the first axiom for an operation defines its existence. The following states that if two stations were connected in the previous network then there is an operation which can return a network where they are disconnected. The guard statement that indicates the stations are connected in the previous network is only for clarification.

```
/* For every two stations A and B that are connected in network N, there */
/* is an operation, disconnect<N, A, B>, which disconnects A and B and   */
/* returns a new network.                                                */
/* This statement contains no constructive content.                      */
NET_Disconnect_identity: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      ((disconnect <N, A, B >) = (disconnect <N, A, B >)))));
```

The *disconnect* operation must be specified to return a new network in which the two stations are disconnected.

```
/* For every two stations A and B that are connected in network N, A and */
/* B are disconnected in the new network returned from the               */
/* disconnect<N, A, B> operation.                                        */
/* This statement contains no constructive content.                      */
```

```
NET_Disconnect_not_connected: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (not IsConnected[(disconnect <N, A, B >), A, B ])) ) );
```

A result of being disconnected is that both of the stations are now not busy in the new network.

```
/* For every two stations A and B that are connected in network N, A and */
/* B are both not busy in the new network returned from the              */
/* disconnect<N, A, B> operation.                                        */
/* This statement contains no constructive content.                      */
NET_Disconnect_not_busy: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      ( (not Busy[(disconnect <N, A, B >), A ]) &
      (not Busy[(disconnect <N, A, B >), B ]) )) ) );
```

As with previous operations, it must be stated that all other connections are maintained and that all previously busy stations are still busy. Once again, the *NET_Disconnect_remains_busy* is not necessary but does save some steps in the proofs.

```
/* For every two stations A and B that are connected in network N, the   */
/* new network returned from the disconnect<N, A, B> operation maintains */
/* all the connection from N except the connection between A and B.      */
/* This statement contains no constructive content.                      */
NET_Disconnect_connections: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (for X, Y being Station holds
        (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
          (IsConnected[N, X, Y ] iff
          IsConnected[(disconnect <N, A, B >), X, Y ])) )) ) );

/* For every two stations A and B that are connected in network N, the   */
/* new network returned from the disconnected<N, A, B> operation will    */
/* maintain the busy state of all stations in N.                         */
/* This statement contains no constructive content.                      */
NET_Disconnect_remains_busy: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (for Z being Station holds
        (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[(disconnect <N, A, B >), Z ])) )) ) );
```

3.2.5 Additional Axioms and Theorems

The previous axioms and primitive operations make up the core of the network model. Other additional axioms were added to further enhance the model with operations that would be required and theorems were introduced to simplify proofs later on. This section will identify the additional axioms and theorems.

Since it has been specified that a busy station is connected to some other station, we need to introduce a method of determining who that station is connected to. This requires the introduction of the *connected_to* operation. This operation will take a busy station in a network and return the station that it is connected to.

```
/* The primitive operation connected_to<N, A> returns the station that   */
/* station A is connected to in network N.                               */
/* This operation is realized by the implemented operation called        */
/* networks-connected_to.                                                */
given connected_to being (<Network, Station > -> Station) rby networks-connected_to;
{<<connected_to:networks-connected_to>> }
```

The next step as with all of the previous operations is to introduce the axioms which specify the characteristics of the operation. The first characteristic is that of identity. The next is that the station returned must be specified so that it is not equal to the original station (this is true since we do not allow stations to connect to themselves). Also, it must be specified that the station returned is the one connected to the original station. This could be separated into three axioms but for convenience they were conjuncted in the *NET_Find_station* axiom.

```
/* For every station A that is busy in network N, there is an operation, */
/* connected_to<N, A>, which returns the station that A is connected to. */
/* This statement contains no constructive content.                      */
NET_Find_station: (for N being Network holds
  (for A being Station holds
    (Busy[N, A ] implies
      ( (A <> (connected_to <N, A >)) & IsConnected[N, A, (connected_to <N, A >) ] &
      ((connected_to <N, A >) = (connected_to <N, A >)) ) ) ) )
```

A data type and axiom which state the existence of an empty network are given to provide the ability to create new networks which have no stations.

```
/* IsEmpty[N] is true if network N contains no stations.                 */
given IsEmpty being [Network];
```

```
/* For every empty network, there are no stations contained in it.      */
/* This axiom is realized by an implemented operation called            */
/* networks-NET_Empty_network.                                          */
NET_Empty_network: (ex N being Network st
  ( IsEmpty[N ] & (for X being Station holds
    (not ContainedIn[N, X ]) ) ) ) rby networks-NET_Empty_network;
{<<networks-NET_Empty_network:networks-NET_Empty_network>>}
```

A theorem, *NET_Equal_network*, proving that for every network there exists a copy of the original network was proven. This was created to handle cases where an operation on the network would just return the original network. The reason that a theorem was proven to return the copy of the network was to satisfy the network state transition model. Another reason is that this theorem shows that membership and connections are maintained. If this was not specified here, it would have to be proven in many different places in proofs that required the original network to be returned. The prefix on the name of the theorem identifies that it is contained in a module called *primitives*. The *rby* statement indicates that this theorem is realized by its proof.

```
/* For every network N, there exists a exact copy of the N which        */
/* maintains membership and connections.                                */
/* This statement contains no constructive content.                     */
primitives.NET_Equal_network: (for N being Network holds
  (ex N' being Network st
    ( (N' = N) & (for X, Y being Station holds
      ( (ContainedIn[N, X ] iff ContainedIn[N', X ]) &
      (ContainedIn[N, Y ] iff ContainedIn[N', Y ]) &
      (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ]) ) ) ) ) )
rby primitives.NET_Equal_network$P;
{<<primitives.NET_Equal_network$P:primitives.NET_Equal_network$P>>}
```

In most operation/feature proofs, it can be seen that all of the axioms of a specific operation (i.e. connect, disconnect, etc.) would have to be used and thus proven to exist. In order to reduce some of the redundancy, separate theorems were proven to show that the characteristics of each of the operations could be conjuncted. Also, the theorems show the existence of the network that is a result of the operation and that it maintains the characteristics defined in the axioms. The following are the theorems.

```
/* Introduce a theorem which proves the existence of the network        */
/* resulting from adding a station to a previous network.  This theorem  */
/* also identifies the characteristics of the add_station operation.     */
primitives.NET_Add_station: (for N being Network holds
  (for A being Station holds
```

```
    ((not ContainedIn[N, A]) implies
      (ex N' being Network st
        ( (N'=(add_station <N, A>)) & ContainedIn[N', A] &
        (for X, Y being Station holds
          (( (X<>A) & (Y<>A) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])))) &
        (for Z being Station holds
          (( ContainedIn[N, Z] or (Z=A) ) iff ContainedIn[N', Z])) )))))
rby primitives.NET_Add_station$P;


/* Introduce a theorem which proves the existence of the network          */
/* resulting from removing a station from a previous network.  This       */
/* theorem also identifies the characteristics of the remove_station      */
/* operation.                                                             */
primitives.NET_Remove_station: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N, A] & (not Busy[N, A])) implies
      (ex N' being Network st
        ( (N'=(remove_station <N, A>)) & (not ContainedIn[N', A]) &
        (for X, Y being Station holds
          (( (X<>A) & (Y<>A) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])))) &
        (for Z being Station holds
          (ContainedIn[N, Z] iff ( ContainedIn[N', Z] or (Z=A) ))) )))))
  rby primitives.NET_Remove_station$P;


/* Introduce a theorem which proves the existence of the network          */
/* resulting from attempting to connect two idle stations of a previous   */
/* network.  This theorem also identifies the characteristics of the */
/* connect operation in this situation.                                   */
primitives.NET_Idle_connect: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A]) & (not Busy[N, B]) ) implies
      (ex N' being Network st
        ( (N'=(connect <N, A, B>)) & IsConnected[N', A, B] &
        (for X, Y being Station holds
          (( (X<>A) & (X<>B) & (Y<>A) & (Y<>B) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])))) &
        (for Z being Station holds
          (( (Z<>A) & (Z<>B) ) implies
            (Busy[N, Z] iff Busy[N', Z])))) )))))
  rby primitives.NET_Idle_connect$P;


/* Introduce a theorem which proves the existence of the network          */
```

```
/* resulting from attempting to connect two stations of a previous       */
/* network where one or both of the stations are busy.  This theorem     */
/* also identifies the characteristics of the connect operation in this  */
/* situation.                                                            */
primitives.NET_Busy_connect: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A] or Busy[N, B] ) implies
      (ex N' being Network st
        ( (N'=N) &
        (for X, Y being Station holds
          (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (Busy[N, Z] iff Busy[N', Z])) )))))
rby primitives.NET_Busy_connect$P;


/* Introduce a theorem which proves the existence of the network         */
/* resulting from disconnecting two stations that were connected in a    */
/* previous network.  This theorem also identifies the characteristics   */
/* of the disconnect operation.                                          */
primitives.NET_Disconnect: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B] implies
      (ex N' being Network st
        ( (N'=(disconnect <N, A, B>)) & (not IsConnected[N', A, B]) &
        (not Busy[N', A]) & (not Busy[N', B]) &
        (for X, Y being Station holds
          (( (X<>A) & (X<>B) & (Y<>A) & (Y<>B) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])))) &
        (for Z being Station holds
          (( (Z<>A) & (Z<>B) ) implies
            (Busy[N, Z] iff Busy[N', Z])))) )))))
  rby primitives.NET_Disconnect$P;
```

Chapter 4

Operation Modelling Method

4.1 Method Description

By evaluating the process by which the network model was developed, a method for specifying an operation can be formed. This method characterizes and develops the operations and features of the network. The following is the method for adding new operations or features to a network model.

1. Specify properties.
2. Identify operations that affect the properties.
3. Provide fundamental specifications of the operations.
4. Implement primitive operations.
5. Prove other properties of operations.

Figure 4.1: Operation Modelling Method

The first step is to specify properties of the operation or feature. Properties of the network and stations must be defined. It is these properties that are used to reason about the state of the network. An example from the network primitives is the introduction of the network and station predicates. This property introduces the network and station data types in the model.

The next step is to identify operations that affect the properties. These are the primitive operations that need to be performed. In this example, the *add_station*, *remove_station*, *connect*, and *disconnect* operation were introduced as primitive operations.

From the operations identified in the last step, the fundamental specifications of the operations must be stated. These are the characteristics of how the operation works. As an example, for the *add_station* operation, four fundamental specifications

were introduced, *NET_Add_station_identity*, *NET_Add_station_contained*, *NET_Add_station_connections*, and *NET_Add_station_remains_same*. This specification identification is the one area that requires external verification.

From these specifications, the implementations of the primitive operations can be derived. This step must be hand-verified by the user to ensure the implementation matches the specifications.

Finally other properties of the operations can be proven. These could possibly derive some new compound operations. In the case of the *add_station* operation, the four fundamental characteristics were used to prove the existence of the network resulting from the *add_station* operation, *NET_Add_station*.

It is this method of operation modelling that is used to incrementally add features (or functionality) to the network model. By using this approach, an evaluation method can be developed to determine when operations on a network cause interactions. This will be discussed in Section 6. The next section will describe the derivation of more complex theorems and the addition of features to the network model.

Chapter 5

Complex Network Operations

5.1 Forced Connection

5.1.1 Motivation

The operation to force a connection in a network is a complex operation which involves many of the basic operations such as connecting, disconnecting, and locating a station. After specifying the network primitives and basic operations, the proof of the forced connection operation was used to satisfy the following goals:

1. Test the network primitives and basic operations to ensure they provide an adequate base network model.

2. Evaluate the steps necessary to prove a complex operation in this model.

Since the forced connection operation uses many of the primitives and basic operations, it assists in verifying the completeness of the current network model. This proof helps to identify missing primitives and incorrect specifications of the primitives. At the same time, the proof was a major stepping stone in understanding if the idea of creating a telephony network model in MIZAR-C was feasible. By evaluating the steps taken to resolve the proof of the forced connection theorem, the techniques required to develop other operations and features could be better understood.

5.1.2 Defining a Forced Connection

The following is the theorem proving the existence of a forced call operation:

```
/* This theorem proves the ability for a connection to be forced between */
/* two stations.  It states that the new network contains the connected  */
/* stations (or it returns the original network in the case that the     */
/* stations do not exist in the network) and that there exists an        */
/* affected network such that for all stations not in the affected       */
/* network, all memberships and connections are maintained.              */
/* It is important to note that this theorem can be proved a number of   */
/* ways.                                                                 */
forced.forced_connection: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2] or (N'=N) ) & (ex Affected being Network st
        (for X, Y being Station holds
          (( (not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y]) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y]))))) ))))
rby forced.forced_connection$P;
```

Figure 5.1: Forced Connection Theorem

This theorem takes a network, *N*, and two stations, *S1* and *S2*. In the case that the stations exist in the network, it returns a network in which the two stations are connected. This function will return the old network in the case that the stations did not exist in the original network. The new network maintains all connections from the old network except those that the given stations were involved in. If the stations do not exist in the network, the old network is returned. In this case all connections are maintained.

A concern in defining the forced call operation was how to state the effect of the operation (i.e. what remains consistent). One method would be to explicitly define the stations involved in the operation and also state that all other stations are unaffected by the operation. Since it is not known how many stations are involved in a particular operation, it would be necessary to prove a number of theorems for each of the possible cases where a different number of stations were involved. If two stations were involved in an operation, a theorem would state that all stations except those two involved would maintain consistency. In the case of three stations a new theorem would have to be created that identified the three stations explicitly, and so on. This would be quite tedious.

Rather than taking the explicit approach, another method was used. This approach identifies the creation of an affected network of stations. This affected network of stations is a subset of the original network. It consists of those stations that were involved in the operation and is developed in the process of proving the operation.

One comment that should be made is that a number of different proofs could satisfy this definition. One way to prove this is to prove that all stations in the network

can be disconnected and thus the two stations given could be connected. Another method is to prove that the two stations are disconnected from any other stations they are connected to and then connect them to each other. All other stations are left alone. This corresponds to different algorithms used in programming a software module (i.e using Quicksort vs. Mergesort). The second approach is the one taken.

This definition could be specified further so that only one proof could be used. This would require identifying which stations are in the affected network under what conditions. Since there are so many different conditions, the definition would be rather complex. For this reason, this definition was left as it is stated above.

5.1.3 Proving the Forced Call Operation

The following is a brief overview of the proof of the Forced Call operation. For a full listing of the proof, refer to Appendix C.

In proving the forced call operation, it was necessary to evaluate a number of cases. The first case to evaluate was whether or not the stations were in the given network.

S1_contained_or_not: (ContainedIn[N, S1] or (not ContainedIn[N, S1]))
by elim[N,S1](NET_Contained_or_not);
S2_contained_or_not: (ContainedIn[N, S2] or (not ContainedIn[N, S2]))
by elim[N,S2](NET_Contained_or_not);

In the case of either or both stations not being in the network, the *NET_Equal_network* function is used to return the original network. Otherwise, if both stations are in the network, the next case to evaluate is whether the two stations are already connected or not.

S1S2_connected_or_not: (IsConnected[N, S1, S2] or (not IsConnected[N, S1, S2]))
by elim[N,S1,S2](NET_IsConnected_or_not);

If the stations were already connected then the *NET_Equal_network* function is used to return the original network since no other actions are necessary. If the stations are not connected then they must be evaluated to determine if they are the same station. In other words a connection is trying to be made from a station to itself and this would defy the definition of consistency.

S1S2_equal_or_not: ((S1 = S2) or (S1 <> S2))
by elim[S1,S2](NET_Station_or_not);

Again, the *NET_Equal_network* function is used to return the original network if the stations are equal. The evaluation of whether the stations are busy or not occurs next if the stations are not the same.

S1_busy_or_not: (Busy[N, S1] or (not Busy[N, S1]))
by elim[N,S1](NET_Busy_or_not);
S2_busy_or_not: (Busy[N1, S2] or (not Busy[N1, S2]))
by elim[N1,S2](NET_Busy_or_not);

If either of the stations are busy, the *NET_Find_station* function is used to find which station the busy station is connected to. Then the *NET_Disconnect* function is used to disconnect those stations. This changes the given busy stations to a not busy state. Once both stations are in an idle state, *NET_Connect* is used to make the connection between the two stations. An outline listing of this proof is contained in Appendix C.

5.1.4 Problems Encountered During Proof

The first and probably the biggest difficulty in this proof was proving that the operation maintained consistency. The basic operations had consistency built into them. Thus, the specification identified the particular stations involved in the operation. The problem encountered in the forced connection proof was that, depending on the state of the original network (i.e. which stations were involved in a connection), there could be a different number of stations involved in the operation. If neither of the given stations were busy then only two stations were involved in the operation. If one was busy then three stations were involved, and so on.

The notion of an affected network was introduced. This affected network contains all stations involved in the operation. These stations were identified from those specified in the basic operations from Section 3.2. The use of this notion ensured that one generic theorem could be created rather than having one theorem for each of the different cases. This also seemed like a good method to incorporate into further complex operations. The specification of maintaining consistency is as follows:

(ex Affected being Network st
(for X, Y being Station holds
(((not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y])) implies
(IsConnected[N, X, Y] iff IsConnected[N', X, Y])));

Figure 5.2: Consistency Specification

A disadvantage to this approach is that you have to tag the path through the proof tree, carrying the affected network as you go. This is due to the fact that the affected network is directly related to the evaluation of the cases in the proof of the operation.

The creation of the affected network is laborious. For every case in which an affected network is needed, an empty network is introduced with the *NET_Empty_*

network function. As stations are identified as being involved in the operation, they must be added to this network. As they are added, it becomes necessary to prove that all the stations that were in the affected network before the addition are still in the network after. At the same time, it is necessary to show that all stations other than those in the affected network held consistency. This specifies the exact stations involved in the operation and that the operation maintains consistency.

Another problem encountered in this proof occurred when certain cases required that multiple operations be performed (i.e. multiple disconnections). Multiple operations require the transition of networks through numerous changes. A mistake made in the first attempt at this proof was that no connection was made between these different transitions.

An example of this mistake occurs if two stations need to be disconnected from their current connections. It is possible to disconnect the first station in the original network and then disconnect the second station in the original network. This produces a resulting network that only has one of the connections disconnected. The correct way of accomplishing the disconnections is to disconnect the first station in the original network then disconnect the second station from the resulting network of the last disconnect.

These types of mistakes are easily detected since they make it impossible to complete the proof. One issue that complicates this problem is if the definition of the theorem is changed to match the proof. In this case, the definition might look similar to the original definition but it would be incorrect. It is important to ensure proper theorem definitions are created based upon specifications and that the proofs follow from the definition.

5.2 Call Attempt

5.2.1 Motivation

Most telephony features are based upon the criteria that a basic call is made and a connection exists between the calling party and the called party. Building the call attempt operation as a separate theorem is necessary to avoid excessively long proofs when developing features. This corresponds to procedure abstraction in regular programming languages. Since all features will hinge upon a call attempt, the operation is abstracted into its own theorem and then it is evaluated in the development of features.

5.2.2 Defining a Call Attempt

The following is the theorem proving the existence of a call attempt operation:

```
/* This theorem proves the existence of a network resulting from a call  */
/* attempt.  The stations will be connected in the new network if they   */
/* were idle in the original network.  The original network is returned  */
/* if either of the stations are busy.  This theorem also introduces     */
/* an affected set.  This affected set will only contain the identified  */
/* stations in the call if they were connected.                          */
basic.basic_call: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2] or (N'=N) ) &
      (ex Affected being Network st
        (for X, Y being Station holds
          (( (not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y]) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (ContainedIn[Affected, Z] implies ( (Z=S1) or (Z=S2) )))) ))))
rby basic.basic_call$P;
```

Figure 5.3: Call Attempt Theorem

This theorem takes a network, *N*, and two stations, *S1* and *S2*. From these, a new network is returned which is either the old network, in the case of the connection not being made, or a copy of the old network with the exception that a new connection exists between the two stations.

This theorem also uses the creation of an affected set in order to define the stations involved in the operation. Unlike the forced connection operation where we did not define the contents of the affected set, it is required for call attempt. This way, it is stated explicitly that everything remains the same except for the stations *S1* and *S2* which are involved in the operation.

For ease of reading the call attempt operation is redefined as follows:

```
/* Introduce a definition of the basic call attempt theorem for ease     */
/* of reading.                                                           */
basic.BasicCall: define basic.BasicCall of N being Network,
A being Station, B being Station, N' being Network by
  ( ( IsConnected[N', A, B] or (N'=N) ) &
    (ex Affected being Network st
      (for X, Y being Station holds
        (( (not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y]) ) implies
          (IsConnected[N, X, Y] iff IsConnected[N', X, Y]))) &
      (for Z being Station holds
        (ContainedIn[Affected, Z] implies ( (Z=S1) or (Z=S2) )))) );
```

Figure 5.4: Call Attempt Redefinition

This allows us to state that *N'* is the result of station *S1* calling station *S2* in network *N* as follows:

```
basic.BasicCall[N, S1, S2, N'];
```

5.2.3 Proving the Call Attempt Operation

In proving the call attempt operation, it is necessary to evaluate a number of cases. These cases are identical to that of the forced connection operation with the exception of how busy stations are handled. If either of the stations are busy, we return the original network. If both of the stations are not busy then *NET_Connect* is used to connect the two stations. An outline listing of this proof is contained in Appendix D.

5.2.4 Problems Encountered During Proof

The problems found in the development of the forced connection operation are evident in the proof of the call attempt operation. Although the creation of the affected networks and proving that consistency is maintained is long and tedious, it will ensure that the proofs of features are simple. This is true since the specification of a feature will state consistency in terms of the call attempt operation.

Chapter 6

Individual Features

The intent of this project was to specify features and hopefully determine a methodology that could be used to ensure no feature interactions occur in the implementations of these features. As discussed in Chapter 1, there are three types of feature interactions. These are *conflict, ambiguity,* and *resource contention.* To limit the size of this project, we focussed on conflicting features.

The features described in this chapter are actually side effects of a call attempt. In other words, in each of these features the definition is based upon a call attempt. If the call attempt succeeds, and there is a connection made, then the feature states are evaluated.

6.1 Call Display

6.1.1 Defining Call Display

The call display feature allows the display of the number of a calling party. The following is a specification of call display:

This feature occurs as a side effect to a connected call. First a call attempt is made. If the connection is made and the called station, *S2*, has the call display feature on then the identifier of the calling station can be retrieved. The *get_id* operation is the required operation for retrieving identifiers of stations. Since this operation does not exist, it will have to be introduced and specified.

6.1.2 Proving the Call Display Feature

The basic outline of the proof for call display is to make a call attempt and if a connection exists and the called party has call display on then get the calling party's identifier. To be able to complete this proof, it was necessary to introduce some changes to the network model. These were introduced based on the method developed in Chapter 4.1. These changes were made as naively as possible. In other words, no knowledge was included about future features. If this knowledge were specified, the feature interaction problems would be minimal.

```
/* This theorem proves that if two stations S1 and S2 from a network N   */
/* are connected using the Basic Call Theorem and S2 has call display on */
/* then the identifier of S1 can be determined.                          */
display.Call_Display: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N', S2] ) implies
        (ex Id being Identifier st
          (Id=(get_id <N', S1>)))) ))))
  rby display.Call_Display$P;
```

Figure 6.1: Call Display Theorem

First, a new property, *Identifier*, had to be added. An Identifier contains information about a station. Then another property was needed to be able to identify the state of the call display feature (either ON or OFF), *DisplayOn*. A decider axiom was introduced to determine the state of call display for a station, *NET_Display_on_or_not*. These and the following predicates and axioms can be seen in Figure 6.2.

```
/* Introduce the Identifier data type. */
given Identifier being [Any ];

/* DisplayOn[N, A] is true if station A has call display on in network N.*/
given DisplayOn being [Network, Station ];

/* The primitive operation get_id<N, A> returns the identifier of the    */
/* station A in network N.                                               */
/* This operation is realized by the implemented operation called        */
/* network-get_id.  The primitive operation must be hand verified to     */
/* ensure it matches its specifications.                                 */
given get_id being (<Network, Station > -> Identifier) rby networks-get_id;
{<<networks-get_id:networks-get_id>> }

/* For every station A, it either has call display on in network N or it */
/* does not.                                                             */
/* This statement is realized by an implemented operation called         */
/* networks-Display_on_or_not.                                           */
NET_Display_on_or_not: (for N being Network holds
  (for A being Station holds
    ( DisplayOn[N, A ] or (not DisplayOn[N, A ]) ) ) )
rby networks-Display_on_or_not;
{<<networks-Display_on_or_not:networks-Display_on_or_not>>}
```

```
/* For every station A and network N there exists an operation,          */
/* get_id<N, A>, which returns the identifier of A.                      */
/* This statement contains no constructive content.                      */
NET_Get_identifier: (for N being Network holds
  (for A being Station holds
    ((get_id <N, A >) = (get_id <N, A >)) ) );
```

Figure 6.2: Network Model Changes for Call Display

The next step of the method was to identify operations that affected these properties. The function, *get_id*, which takes a network and station and returns the identifier of the station was introduced. This was the only operation needed. From this operation, its specification had to be detailed. This resulted in the introduction of the identity axiom, *NET_Get_identifier*.

Note that nothing in this specification signifies that the Identifier is unique. At the same time, no future knowledge of features such as call block are specified. This is the naive approach to specifying a feature.

These specifications identify the new primitive, *get_id*, for the network. This has to be externally implemented based upon the specifications. The function returns the station identifier from the station in the given network. The important point to note is that when creating implementations such as the *get_id* function, there is room for error to occur in MIZAR-C. The implementations must be created based on the specifications in the proofs. From the specification given about the *get_id* function, the implementation should not contain any information regarding the state of features not yet specified.

The final step in the method was to prove other properties of the operations. In this case, the actual theorem defined in Section 6.1.1 needed to be proven. The proof of this theorem is quite simple:

- Prove the existence of the call attempt operation.
- Prove that if the two stations get connected and the called station has call display on then the identifier can be determined.

The complete listing of this proof is contained in Appendix E.

6.2 Call Block

6.2.1 Defining Call Block

Call block is a privacy feature that ensures that if a caller has the feature on then any called party with call display should not be able to get the caller's identifier. The following is a specification of call block:

```
/* This theorem proves that if two stations S1 and S2 from a network N,  */
/* are connected using the Basic Call Theorem, S2 has call display on,   */
/* and S1 has call block on then the identifier of S1 is the unknown     */
/* identifier.                                                           */
block.Call_Block: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N, S2] & BlockOn[N, S1] ) implies
        (ex Id being Identifier st
          (Id=UnknownId))) ))))
rby block.Call_Block$P;
```

Figure 6.3: Call Block Theorem

This feature takes a network, *N*, and two stations, *S1* and *S2*, and attempts to make the connection. The network returned is defined by the call attempt operation. If the connection is made and the calling station, *S1*, has call block on and the called station, *S2*, has call display on then the identifier of the calling station is returned as unknown.

6.2.2 Proving the Call Block Feature

The proof of call block first instantiates a call attempt. If the connection exists, the calling party has call block on and the called party has call display on then the identifier of the calling party is unknown. In order to complete this proof changes to the network model were needed. These changes were also made without being based on any knowledge of future features.

Once again, using the method defined in Chapter 4.1 we first define new properties. A new data type was introduced, *UnknownId*, which signifies a universal unknown identifier. A predicate, *BlockOn*, and decider, *NET_Block_on_or_not*, were included for determining if the call block feature was on. These can be seen in Figure 6.4.

```
/* Introduce the unknown identifier which is of type Identifier.         */
given UnknownId being Identifier;

/* BlockOn[N, A] is true if station A has call block on in network N.    */
given BlockOn being [Network, Station ];

/* For every station A, it either has call block on in network N or it   */
/* does not.                                                             */
/* This statement is realized by an implemented operation called         */
/* network-Block_on_or_not.                                              */
```

```
NET_Block_on_or_not: (for N being Network holds
  (for A being Station holds
    ( BlockOn[N, A ] or (not BlockOn[N, A ]) ) ) )
rby networks-Block_on_or_not;
{<<networks-Block_on_or_not:networks-Block_on_or_not>>}

/* For every station A that has call block on in network N, the          */
/* identifier returned from the get_id<N, A> operation is equal to the   */
/* unknown identifier.                                                   */
/* This statement contains no constructive content.                      */
NET_get_id_block_on: (for N being Network holds
  (for A being Station holds
    (BlockOn[N, A ] implies ((get_id <N, A >) = UnknownId)) ) );
```

Figure 6.4: Network Model Changes for Call Block

The next step in the method requires identifying operations that affect the properties. In this case, we can see that the *get_id* operation is required. This is the first indication that a feature interaction may exist.

Next, we identify axioms that specify the operation. In this case, the specifications that already exist are kept and a new one is added. A new axiom was needed to state that if a calling party has call block on then the *get_id* function would return with an unknown identifier. This verifies that a feature interaction exists since the specification of an existing operation is modified. This is also the start of the resolution of the feature interaction.

Since we have modified the specification of the operation, the implementation of the operation will have to be modified. These modifications will take into account the new or changed specification. It again must be stressed that the user must ensure that the implementations correlate with the specifications that are defined. If they do not, there is no guarantee that the feature interaction will be resolved. The last step of the method is to prove the theorem about call block that was defined in Section 6.2.1.

At this point it became necessary to specify the identifiers of stations even further. Particularly, it needed to be stated that the actual identifiers of stations were unique and meaningful. A unique identifier occurs when the calling station does not have call block on. Therefore, when the feature is off, *get_id* returns an identifier which is not equal to the unknown identifier. A meaningful identifier is one which can be used to return the station assigned to a unique identifier. This required the introduction of a new function, *get_station*, which takes a network and identifier and returns the corresponding station.

```
/* The primitive operation get_station<N, I> returns the station          */
/* associated with the identifier I in network N.                         */
/* This operation is realized by the implemented operation called         */
/* networks-get_station.  This primitive operation must be hand-verified */
/* to ensure it matches its specifications.                               */
given get_station being (<Network, Identifier > -> Station) rby networks-get_station;
{<<networks-get_station:networks-get_station>> }

/* For every station A that does not have call block on in network N,     */
/* the identifier returned from the get_id<N, A> operation is not equal   */
/* to the unknown identifier.                                             */
/* This statement contains no constructive content.                       */
NET_Unique_identifier: (for N being Network holds
  (for A being Station holds
    ((not BlockOn[N, A ]) implies ((get_id <N, A >) <> UnknownId)) ) );

/* For every station A and network N, the get_id<N, A> operation will     */
/* return an identifier I such that the get_station<N, I> operation       */
/* returns the station A.                                                 */
/* This statement contains no constructive content.                       */
NET_Meaningful_identifier: (for N being Network holds
  (for A being Station holds
    ((get_station <N, (get_id <N, A >) >) = A) ) );
```

Figure 6.5: Network Model Changes for Unique Identifiers

6.3 Feature Interaction Prevention Method

From the experience of formalizing features, a method for detection and resolution of feature interactions was developed. It is based on the original operation modelling method with some modifications. The following is the feature interaction prevention method:

1. Specify properties.

2. Identify operations that affect the properties. If an operation is used that was specified previously, there could be a feature interaction(s). This along with the next step provide the method of detecting possible feature interactions.

3. Provide fundamental specifications of the operations. If new specifications of an existing operation are introduced, there is some form of a feature interaction that could occur.

4. Implement or re-implement operations based on the specifications. Modifications of existing operations to match old and new specifications will ensure that

any possible feature interactions are resolved in the network model specifications.

5. Prove other properties of operations.

Figure 6.6: Feature Interaction Prevention Method

Chapter 7

Conclusions

7.1 Feature Interaction Specifications in MIZAR-C

In this work, we have shown that feature interactions can be specified using MIZAR-C and the following method to ensure that no interactions will occur in the running implementation.

1. Specify properties.
2. Identify operations that affect the properties.
3. Provide fundamental specifications of the operations.
4. Implement or re-implement operations based on the specifications.
5. Prove other properties of operations.

Figure 7.1: Feature Interaction Prevention Method

This approach resolves feature interactions in specifications in two important steps:

1. Detecting possible feature interactions during the specification of operations. This is accomplished in steps 2 and 3 of the above method.
2. Resolving the feature interactions by modifying implemented operations. This is accomplished in step 4 of the above method.

Feature interactions are detected in the network specifications when the axioms introduced interact with existing axioms or primitive operations. If changes must be made to the existing operation specifications then some form of feature interaction is possible. An example occurs in the call block function when the axiom is introduced that when call block is on, a station's identifier is equal to unknown. This axiom

affects the operation of the primitive operation *get_id*. Without changing the implementation of this function from what was implemented for call display, the function will always return the actual identifier of a station. This contradicts what the call block feature specifies. This problem is identified when we are unable to complete the proof of the feature.

The resolution of these feature interactions is accomplished by modifying the primitive operations of the network model. The operations must be re-implemented to reflect all the characteristics specified in the network model. In the example given it is the *get_id* operation that is effected by the additional axioms, therefore it is necessary to re-implement this operation.

7.2 Contributions to Research

This is a new approach to dealing with the feature interaction problem. Other research tends to focus on the detection of interactions during system execution or simulations. That approach is not guaranteed to resolve all features interactions. In this new approach, the specification of the network model ensures that all feature interactions are detected and resolved prior to system execution.

Another benefit to this approach is that once the specification is complete, MIZAR-C automates the creation of the implementation of the full system. The automated use of constructive proofs for programming simplifies the problem of converting logic to programming constructs.

Overall, resolving all feature interactions in the specification reduces the overhead of detecting them in the execution or simulation. This corresponds to common software engineering practices. The sooner the errors are detected, the less the cost associated with their resolution. Since this approach detects all feature interactions during specification, there should not be any overhead for resolution in the executable.

7.3 Future Work

The necessity to implement primitive operations in MIZAR-C hinders the level of correctness available for the specifications. This requires that the user ensure that all implementations are formed based upon the specifications of the primitive operations. For MIZAR-C to provide a higher level of correctness, it would be necessary to eliminate primitive implementations all together. It may also be necessary to provide a more mathematical foundation.

The problem of moving to a stricter formal system is the time required to prove all aspects of the primitive operations. If it becomes necessary to specify and verify these implementations, much more time will be spent on verifying new features and possibly modifying old features. A question that needs to be answered is how much specification and verification is needed for the network model. If only a small level of correctness is necessary then the MIZAR-C approach is very effective. If a high level

is needed then some other system would be required.

One of the most difficult aspects of this project was trying to state that when an operation occurs in a network, all characteristics of the network, except those involved in the operation, remain the same. MIZAR-C has no easy method of dealing with this problem. A resolution to this may be to introduce a construct in the MIZAR-C system which gives the user the ability to declare that everything remains the same except for a list for changes caused by the operation. This approach is done in the Prototype Verification System (PVS) [5] using the PVS WITH construct and in VDM [12] using the VDM bar ($^-$) notation.

The extracted code that MIZAR-C generates is very simply a set of code that would be useful for simulations or the continuation of building proofs involving the network model. We would not be able to use the generated code in an actual executing telephony system for the simple fact that the code is not optimized. An interesting problem with the MIZAR-C tool would be to attempt to develop a method of creating optimized code.

Another problem found with using the MIZAR-C system is that modelling of events is difficult. An event corresponds to an activity such as a flashhook signal for triggering feature actions. These events can occur at undetermined times. This "random" activity is difficult to formalize.

One method of handling this situation would be to model the event as a data type. Then we could identify the occurrence of an event in a network from a particular station similar to the way we specify that a station can occur in a network. Although this is not the preferred method of modelling an event, it would allow a specification to be built. The specifications could be built such that given an event, certain actions could take place. To evaluate the feasibility of this approach, the features requiring events would have to be modelled.

The modelling of resource contention requires the ability to enumerate. Resource contention occurs when the demand for a resource is greater than the number of resources available. Enumerating resources can be accomplished in MIZAR-C through the use of finite sequences. Finite sequences are discussed in Kathy Kippen's thesis [14].

Modelling telephony networks in MIZAR-C is effective in attaining a low level of correctness. As long as implementations are developed based on the specifications, this correctness is maintained. The avoidance of implementations and possibly the introduction of a mathematical foundation may provide a higher level of correctness but may also slow the process down.

Appendix A

Network and Station Extensions

The following information about the basic Network and Station data types has been added to Mizar-C:

given Station being [Any];

given Network being [Any];

given Identifier being [Any];

given UnknownId being Identifier;

given Busy being [Network, Station];

given IsConnected being [Network, Station, Station];

given ContainedIn being [Network, Station];

given IsEmpty being [Network];

given DisplayOn being [Network, Station];

given BlockOn being [Network, Station];

given add_station being (<Network, Station > -> Network) rby networks-add_station;
`{<<networks-add_station:networks-add_station>> }`

given remove_station being (<Network, Station > -> Network) rby networks-remove_station;
`{<<networks-remove_station:networks-remove_station>> }`

given connect being (<Network, Station, Station > -> Network) rby networks-connect;
`{<<networks-connect:networks-connect>> }`

given disconnect being (<Network, Station, Station > -> Network) rby networks-disconnect;

```
{<<networks-disconnect:networks-disconnect>> }
```

given connected_to being (<Network, Station > -> Station) rby networks-connected_to;
```
{<<networks-connected\_to:networks-connected\_to>> }
```

given get_id being (<Network, Station > -> Identifier) rby networks-get_id;
```
{<<networks-get_id:networks-get_id>> }
```

given get_station being (<Network, Identifier > -> Station) rby networks-get_station;
```
{<<networks-get_station:networks-get_station>> }
```

NET_Station_or_not: (for A being Station holds
(for B being Station holds
((A = B) or (A <> B))))
rby networks-Station_or_not;
```
{<<networks-Station_or_not:networks-Station_or_not>>}
```

NET_Busy_or_not: (for N being Network holds
(for A being Station holds
(Busy[N, A] or (not Busy[N, A]))))
rby networks-Busy_or_not;
```
{<<networks-Busy_or_not:networks-Busy_or_not>>}
```

NET_IsConnected_or_not: (for N being Network holds
(for A, B being Station holds
(IsConnected[N, A, B] or (not IsConnected[N, A, B]))))
rby networks-IsConnected_or_not;
```
{<<networks-IsConnected_or_not:networks-IsConnected_or_not>>}
```

NET_Contained_or_not: (for N being Network holds
(for A being Station holds
(ContainedIn[N, A] or (not ContainedIn[N, A]))))
rby networks-Contained_or_not;
```
{<<networks-Contained_or_not:networks-Contained_or_not>>}
```

NET_Display_on_or_not: (for N being Network holds
(for A being Station holds
(DisplayOn[N, A] or (not DisplayOn[N, A]))))
rby networks-Display_on_or_not;
```
{<<networks-Display_on_or_not:networks-Display_on_or_not>>}
```

NET_Block_on_or_not: (for N being Network holds
(for A being Station holds
(BlockOn[N, A] or (not BlockOn[N, A]))))
rby networks-Block_on_or_not;
```
{<<networks-Block_on_or_not:networks-Block_on_or_not>>}
```

NET_IsConnected_Busy: (for N being Network holds
(for A, B being Station holds
(IsConnected[N, A, B] implies
(Busy[N, A] & Busy[N, B]))));

NET_Add_station_identity: (for N being Network holds
(for A being Station holds
((not ContainedIn[N, A]) implies
((add_station <N, A >) = (add_station <N, A >)))));

NET_Add_station_contained: (for N being Network holds
(for A being Station holds
((not ContainedIn[N, A]) implies
ContainedIn[(add_station <N, A >), A])));

NET_Add_station_connections: (for N being Network holds
(for A being Station holds
((not ContainedIn[N, A]) implies
(for X, Y being Station holds
(((X <> A) & (Y <> A)) implies
(IsConnected[N, X, Y] iff IsConnected[(add_station <N, A >), X, Y]))))));

NET_Add_station_remains_same: (for N being Network holds
(for A being Station holds
((not ContainedIn[N, A]) implies
(for Z being Station holds
((ContainedIn[N, Z] or (Z = A)) iff ContainedIn[(add_station <N, A >), Z])))))
rby networks-Add_station_remains_same;
`{<<networks-Add_station_remains_same:networks-Add_station_remains_same>>}`

NET_Remove_station_identity: (for N being Network holds
(for A being Station holds
((ContainedIn[N, A] & (not Busy[N, A])) implies
((remove_station <N, A >) = (remove_station <N, A >)))));

NET_Remove_station_not_contained: (for N being Network holds
(for A being Station holds
((ContainedIn[N, A] & (not Busy[N, A])) implies (not ContainedIn[(remove_station <N, A >), A]
`{<<NET_Remove_station_not_contained$P31:R-NONE>>}`

NET_Remove_station_connections: (for N being Network holds
(for A being Station holds
((ContainedIn[N, A] & (not Busy[N, A])) implies (for X, Y being Station holds
(((X <> A) & (Y <> A)) implies
(IsConnected[N, X, Y] iff IsConnected[(remove_station <N, A >), X, Y]))))));

NET_Remove_station_remains_same: (for N being Network holds
(for A being Station holds
((ContainedIn[N, A] & (not Busy[N, A])) implies
(for Z being Station holds
(ContainedIn[N, Z] iff (ContainedIn[(remove_station <N, A >), Z] or (Z = A)))))))
rby networks-Remove_station_remains_same;
`{<<networks-Remove_station_remains_same:networks-Remove_station_remains_same>>}`

NET_Remove_add_station: (for N being Network holds
(for A being Station holds
((not ContainedIn[N, A]) implies
(N = (remove_station <(add_station <N, A >), A >)))));

NET_Add_remove_station: (for N being Network holds
(for A being Station holds
(ContainedIn[N, A] implies
(N = (add_station <(remove_station <N, A >), A >)))));

NET_Idle_connect_identity: (for N being Network holds
(for A, B being Station holds
(((not Busy[N, A]) & (not Busy[N, B])) implies
((connect <N, A, B >) = (connect <N, A, B >)))));

NET_Idle_connect_is_connected: (for N being Network holds
(for A, B being Station holds
(((not Busy[N, A]) & (not Busy[N, B])) implies
IsConnected[(connect <N, A, B >), A, B])));
`{<<NET_Idle_connect_is_connected$P48:R-NONE>>}`

NET_Idle_connect_connections: (for N being Network holds
(for A, B being Station holds
(((not Busy[N, A]) & (not Busy[N, B])) implies
(for X, Y being Station holds
(((X <> A) & (X <> B) & (Y <> A) & (Y <> B)) implies
(IsConnected[N, X, Y] iff IsConnected[(connect <N, A, B >), X, Y]))))));

NET_Idle_connect_remains_busy: (for N being Network holds
(for A, B being Station holds
(((not Busy[N, A]) & (not Busy[N, B])) implies
(for Z being Station holds
(((Z <> A) & (Z <> B)) implies
(Busy[N, Z] iff Busy[(connect <N, A, B >), Z]))))));

NET_Busy_connect_identity: (for N being Network holds
(for A, B being Station holds

```
    (( Busy[N, A ] or Busy[N, B ] ) implies
      ((connect <N, A, B >) = N)) ) );

NET_Busy_connect_connections: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies
      (for X, Y being Station holds
        (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ]) )) ) );

NET_Busy_connect_remains_busy: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies
      (for Z being Station holds
        (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ]) )) ) );

NET_Disconnect_identity: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      ((disconnect <N, A, B >) = (disconnect <N, A, B >))) ) );

NET_Disconnect_not_connected: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (not IsConnected[(disconnect <N, A, B >), A, B ])) ) );

NET_Disconnect_not_busy: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      ( (not Busy[(disconnect <N, A, B >), A ]) &
      (not Busy[(disconnect <N, A, B >), B ]) )) ) );

NET_Disconnect_connections: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (for X, Y being Station holds
      (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[(disconnect <N, A, B >), X, Y ])) )) ) );

NET_Disconnect_remains_busy: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (for Z being Station holds
        (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[(disconnect <N, A, B >), Z ])) )) ) );

NET_Find_station: (for N being Network holds
```

```
   (for A being Station holds
     (Busy[N, A ] implies
       (ex B being Station st
         ( (A <> B) & IsConnected[N, A, B ] &
         (B = (connected_to <N, A >)) ) )) ) )
rby networks-NET_Find_station;
{<<networks-NET_Find_station:networks-NET_Find_station>>}

NET_Stations_equal: (for N being Network holds
   (for A, B being Station holds
     ((A = B) implies (Busy[N, A] iff Busy[N, B]))));

NET_Empty_network: (ex N being Network st
   ( IsEmpty[N ] &
   (for X being Station holds
     (not ContainedIn[N, X ]) ) ) )
rby networks-NET_Empty_network;
{<<networks-NET_Empty_network:networks-NET_Empty_network>>}

NET_Get_identifier: (for N being Network holds
   (for A being Station holds
     ((get_id <N, A >) = (get_id <N, A >)) ) );

NET_get_id_block_on: (for N being Network holds
   (for A being Station holds
     (BlockOn[N, A ] implies ((get_id <N, A >) = UnknownId)) ) );

NET_Unique_identifier: (for N being Network holds
   (for A being Station holds
     ((not BlockOn[N, A ]) implies ((get_id <N, A >) <> UnknownId)) ) );

NET_Meaningful_identifier: (for N being Network holds
   (for A being Station holds
     ((get_station <N, (get_id <N, A >) >) = A) ) );
```

Appendix B

Primitive Operations

B.1 Specification of Primitive Operations

```
primitives.NET_Add_station: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A]) implies
      (ex N' being Network st
        ( (N'=(add_station <N, A>)) & ContainedIn[N', A] &
        (for X, Y being Station holds
          (( (X<>A) & (Y<>A) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (( ContainedIn[N, Z] or (Z=A) ) iff ContainedIn[N', Z])) )))))
rby primitives.NET_Add_station$P;

primitives.NET_Remove_station: (for N being Network holds
  (for A being Station holds
    ((ContainedIn[N, A] & (not Busy[N, A])) implies
      (ex N' being Network st
        ( (N'=(remove_station <N, A>)) & (not ContainedIn[N', A]) &
        (for X, Y being Station holds
          (( (X<>A) & (Y<>A) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (ContainedIn[N, Z] iff ( ContainedIn[N', Z] or (Z=A) )) )))))
rby primitives.NET_Remove_station$P;

primitives.NET_Idle_connect: (for N being Network holds
  (for A, B being Station holds
    (( (not Busy[N, A]) & (not Busy[N, B]) ) implies
      (ex N' being Network st
        ( (N'=(connect <N, A, B>)) & IsConnected[N', A, B] &
        (for X, Y being Station holds
```

```
            (( (X<>A) & (X<>B) & (Y<>A) & (Y<>B) ) implies
              (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
          (for Z being Station holds
            (( (Z<>A) & (Z<>B) ) implies
              (Busy[N, Z] iff Busy[N', Z])) ))))
rby primitives.NET_Idle_connect$P;

primitives.NET_Busy_connect: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A] or Busy[N, B] ) implies
      (ex N' being Network st
        ( (N'=N) &
        (for X, Y being Station holds
          (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (Busy[N, Z] iff Busy[N', Z])) )))))
rby primitives.NET_Busy_connect$P;

primitives.NET_Disconnect: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B] implies
      (ex N' being Network st
        ( (N'=(disconnect <N, A, B>)) & (not IsConnected[N', A, B]) &
        (not Busy[N', A]) & (not Busy[N', B]) &
        (for X, Y being Station holds
          (( (X<>A) & (X<>B) & (Y<>A) & (Y<>B) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (( (Z<>A) & (Z<>B) ) implies
            (Busy[N, Z] iff Busy[N', Z])) )))))
rby primitives.NET_Disconnect$P;
```

B.2 Proof of NET_Add_Station

```
Add_Station: now
  let N be Network;
  {<<primitives$R111:primitives$R111>> }

  let A be Station;
  {<<primitives$R112:primitives$R112>> }
  now
    assume 10: (not ContainedIn[N, A ]);
    {<<primitives$R113:primitives$R113>>}
```

```
/* Since station A is not contained in network N, we can state the*/
/* characteristics of the add_station operation.                  */

20: ((add_station <N, A >) = (add_station <N, A >))
by elim[N,A](NET_Add_station_identity, _PREVIOUS);
30: ContainedIn[(add_station <N, A >), A ]
by elim[N,A](NET_Add_station_contained, 10);
40: (for X, Y being Station holds
  (( (X <> A) & (Y <> A) ) implies
    (IsConnected[N, X, Y ] iff IsConnected[(add_station <N, A >), X, Y ])) )
by elim[N,A](NET_Add_station_connections, 10);
50: (for Z being Station holds
  (( ContainedIn[N, Z ] or (Z = A) ) iff ContainedIn[(add_station <N, A >), Z ]) )
by elim[N,A](NET_Add_station_remains_same, 10);
{<<primitives$R122:(R-APPLY (R-APPLY networks-Add_station_remains_same
primitives$R111 :tag '$ELIM2120) primitives$R112 :tag '$ELIM2121)>>}
( ((add_station <N, A >) = (add_station <N, A >)) &
ContainedIn[(add_station <N, A >), A ] &
(for X, Y being Station holds
  (( (X <> A) & (Y <> A) ) implies
    (IsConnected[N, X, Y ] iff IsConnected[(add_station <N, A >), X, Y ])) ) &
(for Z being Station holds
  (( ContainedIn[N, Z ] or (Z = A) ) iff ContainedIn[(add_station <N, A >), Z ]) ) )
by conj(20, 30, 40, 50);
{<<primitives$R126:primitives$R122>>}

/* Introduce a new network which represents the results of the    */
/* add_station operation.                                         */

thus (ex N' being Network st
  ( (N' = (add_station <N, A >)) & ContainedIn[N', A ] &
  (for X, Y being Station holds
    (( (X <> A) & (Y <> A) ) implies
      (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
  (for Z being Station holds
    (( ContainedIn[N, Z ] or (Z = A) ) iff ContainedIn[N', Z ]) ) ) )
by exintro(_PREVIOUS);
{<<primitives$R131:(R-LIST (LAMBDA () (FUNCALL networks-add_station
 (R-TUPLE primitives$R111 primitives$R112))) (LAMBDA ()
primitives$R126))>>}

end;
{<<primitives$R132:(R-LIST (LAMBDA () (FUNCALL networks-add_station
 (R-TUPLE primitives$R111 primitives$R112))) (LAMBDA ()
primitives$R126))>>}
```

```
thus ((not ContainedIn[N, A ]) implies
  (ex N' being Network st
    ( (N' = (add_station <N, A >)) & ContainedIn[N', A ] &
    (for X, Y being Station holds
      (( (X <> A) & (Y <> A) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
    (for Z being Station holds
      (( ContainedIn[N, Z ] or (Z = A) ) iff ContainedIn[N', Z ]) ) ) ))
  by direct(_PREVIOUS);
  {<<primitives$R137:primitives$R132>>}

end;
{<<primitives$R138:(LAMBDA (primitives$R111) (LAMBDA (primitives$R112)
 primitives$R132))>>}

NET_Add_station: (for N being Network holds
  (for A being Station holds
    ((not ContainedIn[N, A ]) implies
      (ex N' being Network st
        ( (N' = (add_station <N, A >)) & ContainedIn[N', A ] &
        (for X, Y being Station holds
          (( (X <> A) & (Y <> A) ) implies
            (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
        (for Z being Station holds
          (( ContainedIn[N, Z ] or (Z = A) ) iff ContainedIn[N', Z ]) ) ) )) ) )
    by direct(_PREVIOUS);
    {<<primitives.NET_Add_station$P:primitives$R138>>}
```

B.3 Proof of NET_Remove_Station

```
Remove_Station: now
  let N be Network;
  {<<primitives$R145:primitives$R145>> }

  let A be Station;
  {<<primitives$R146:primitives$R146>> }

  now
    assume 10: ContainedIn[N, A ] & (not Busy[N, A]);
    {<<primitives$R147:primitives$R147>>}

    /* Since station A is contained in network N, we can state the    */
    /* characteristics of the remove_station operation.               */
```

```
20: ((remove_station <N, A >) = (remove_station <N, A >))
by elim[N,A](NET_Remove_station_identity, _PREVIOUS);
30: (not ContainedIn[(remove_station <N, A >), A ])
by elim[N,A](NET_Remove_station_not_contained, 10);
40: (for X, Y being Station holds
   (( (X <> A) & (Y <> A) ) implies
      (IsConnected[N, X, Y ] iff IsConnected[(remove_station <N, A >), X, Y ])) )
by elim[N,A](NET_Remove_station_connections, 10);
50: (for Z being Station holds
   (ContainedIn[N, Z ] iff ( ContainedIn[(remove_station <N, A >), Z ] or (Z = A) )) )
by elim[N,A](NET_Remove_station_remains_same, 10);
{<<primitives$R156:(R-APPLY (R-APPLY networks-Remove_station_remains_same
primitives$R145 :tag '$ELIM2154) primitives$R146 :tag '$ELIM2155)>>}
( ((remove_station <N, A >) = (remove_station <N, A >)) &
(not ContainedIn[(remove_station <N, A >), A ]) &
(for X, Y being Station holds
   (( (X <> A) & (Y <> A) ) implies
      (IsConnected[N, X, Y ] iff IsConnected[(remove_station <N, A >), X, Y ])) ) &
(for Z being Station holds
   (ContainedIn[N, Z ] iff ( ContainedIn[(remove_station <N, A >), Z ] or (Z = A) )) ) )
by conj(20, 30, 40, 50);
{<<primitives$R160:primitives$R156>>}

/* Introduce a new network which represents the results of the    */
/* remove_station operation.                                      */

thus (ex N' being Network st
   ( (N' = (remove_station <N, A >)) & (not ContainedIn[N', A ]) &
   (for X, Y being Station holds
      (( (X <> A) & (Y <> A) ) implies
         (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
   (for Z being Station holds
      (ContainedIn[N, Z ] iff ( ContainedIn[N', Z ] or (Z = A) )) ) ) )
by exintro(_PREVIOUS);
{<<primitives$R165:(R-LIST (LAMBDA () (FUNCALL networks-remove_station
(R-TUPLE primitives$R145 primitives$R146))) (LAMBDA ()
primitives$R160))>>}

end;
{<<primitives$R166:(R-LIST (LAMBDA () (FUNCALL networks-remove_station
(R-TUPLE primitives$R145 primitives$R146))) (LAMBDA ()
primitives$R160))>>}

thus ((ContainedIn[N, A ] & (not Busy[N, A])) implies
   (ex N' being Network st
      ( (N' = (remove_station <N, A >)) & (not ContainedIn[N', A ]) &
```

```
        (for X, Y being Station holds
          (( (X <> A) & (Y <> A) ) implies
            (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
        (for Z being Station holds
          (ContainedIn[N, Z ] iff ( ContainedIn[N', Z ] or (Z = A) )) ) ) ))
    by direct(_PREVIOUS);
    {<<primitives$R171:primitives$R166>>}

end;
{<<primitives$R172:(LAMBDA (primitives$R145) (LAMBDA (primitives$R146)
primitives$R166))>>}

NET_Remove_station: (for N being Network holds
    (for A being Station holds
      ((ContainedIn[N, A ] & (not Busy[N, A])) implies
        (ex N' being Network st
          ( (N' = (remove_station <N, A >)) & (not ContainedIn[N', A ]) &
          (for X, Y being Station holds
            (( (X <> A) & (Y <> A) ) implies
              (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
          (for Z being Station holds
            (ContainedIn[N, Z ] iff ( ContainedIn[N', Z ] or (Z = A) )) ) ) )) ) )
by direct(_PREVIOUS);
{<<primitives.NET_Remove_station$P:primitives$R172>>}
```

B.4 Proof of NET_Idle_Connect

```
Idle_Connect: now
    let N be Network;
    {<<primitives$R179:primitives$R179>> }

    let A, B be Station;
    {<<primitives$R180:primitives$R180>> <<primitives$R181:primitives$R181>> }

    now
      assume 10: ( (not Busy[N, A ]) & (not Busy[N, B ]) );
      {<<primitives$R182:primitives$R182>>}

      /* Since stations A and B are not Busy in network N, we can state */
      /* the characteristics of the connect operation for idle stations.*/

      20: ((connect <N, A, B >) = (connect <N, A, B >))
      by elim[N,A,B](NET_Idle_connect_identity, _PREVIOUS);
      30: IsConnected[(connect <N, A, B >), A, B ]
```

```
by elim[N,A,B](NET_Idle_connect_is_connected, 10);
40: (for X, Y being Station holds
  (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
    (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ])) )
by elim[N,A,B](NET_Idle_connect_connections, 10);
50: (for Z being Station holds
  (( (Z <> A) & (Z <> B) ) implies
    (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ])) )
by elim[N,A,B](NET_Idle_connect_remains_busy, 10);
( ((connect <N, A, B >) = (connect <N, A, B >)) &
IsConnected[(connect <N, A, B >), A, B ] &
(for X, Y being Station holds
  (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
    (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ])) ) &
(for Z being Station holds
  (( (Z <> A) & (Z <> B) ) implies
    (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ])) ) )
by conj(20, 30, 40, 50);

/* Introduce a new network which represents the results of the    */
/* connect_station operation for idle stations.                   */

thus (ex N' being Network st
  ( (N' = (connect <N, A, B >)) & IsConnected[N', A, B ] &
  (for X, Y being Station holds
    (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
      (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
  (for Z being Station holds
    (( (Z <> A) & (Z <> B) ) implies
      (Busy[N, Z ] iff Busy[N', Z ])) ) ) )
by exintro(_PREVIOUS);
{<<primitives$R198:(LAMBDA () (FUNCALL networks-connect (R-TUPLE
primitives$R179 primitives$R180 primitives$R181))) >>}

end;
{<<primitives$R199:(LAMBDA () (FUNCALL networks-connect (R-TUPLE
primitives$R179 primitives$R180 primitives$R181))) >>}

thus (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
  (ex N' being Network st
    ( (N' = (connect <N, A, B >)) & IsConnected[N', A, B ] &
    (for X, Y being Station holds
      (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
    (for Z being Station holds
      (( (Z <> A) & (Z <> B) ) implies
```

```
            (Busy[N, Z ] iff Busy[N', Z ])) ) ) ))
    by direct(_PREVIOUS);
    {<<primitives$R204:primitives$R199>>}

  end;
  {<<primitives$R205:(LAMBDA (primitives$R179) (LAMBDA (primitives$R180)
  (LAMBDA (primitives$R181) primitives$R199)))>>}

  NET_Idle_connect: (for N being Network holds
    (for A, B being Station holds
      (( (not Busy[N, A ]) & (not Busy[N, B ]) ) implies
        (ex N' being Network st
          ( (N' = (connect <N, A, B >)) & IsConnected[N', A, B ] &
          (for X, Y being Station holds
            (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
              (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
          (for Z being Station holds
            (( (Z <> A) & (Z <> B) ) implies
              (Busy[N, Z ] iff Busy[N', Z ])) ) ) )) ) )
  by direct(_PREVIOUS);
  {<<primitives.NET_Idle_connect$P:primitives$R205>>}
```

B.5 Proof of NET_Busy_Connect

```
  Busy_Connect: now
    let N be Network;
    {<<primitives$R248:primitives$R248>> }

    let A, B be Station;
    {<<primitives$R249:primitives$R249>> <<primitives$R250:primitives$R250>> }

    now
      assume 10: ( Busy[N, A ] or Busy[N, B ] );
      {<<primitives$R251:primitives$R251>>}

      /* Since station A or B is Busy in network N, we can state the    */
      /* characteristics of the connect operation for busy stations.    */

      20: ((connect <N, A, B >) = N)
      by elim[N,A,B](NET_Busy_connect_identity, _PREVIOUS);
      30: (for X, Y being Station holds
        (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ]) )
      by elim[N,A,B](NET_Busy_connect_connections, 10);
      40: (for Z being Station holds
```

```
      (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ]) )
    by elim[N,A,B](NET_Busy_connect_remains_busy, 10);
    ( ((connect <N, A, B >) = N) &
    (for X, Y being Station holds
      (IsConnected[N, X, Y ] iff IsConnected[(connect <N, A, B >), X, Y ]) ) &
    (for Z being Station holds
      (Busy[N, Z ] iff Busy[(connect <N, A, B >), Z ]) ) )
    by conj(20, 30, 40);

    /* Introduce a new network which represents the results of the    */
    /* connect operation for busy stations.                           */

    thus (ex N' being Network st
      ( (N' = N) &
      (for X, Y being Station holds
        (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ]) ) &
      (for Z being Station holds
        (Busy[N, Z ] iff Busy[N', Z ]) ) ) )
    by exintro(_PREVIOUS);
    {<<primitives$R266:(LAMBDA () (FUNCALL networks-connect (R-TUPLE
    primitives$R248 primitives$R249 primitives$R250))) >>}

  end;
  {<<primitives$R267:(LAMBDA (primitives$R251) (LAMBDA () (FUNCALL
   networks-connect (R-TUPLE primitives$R248 primitives$R249
  primitives$R250))) )>>}

  thus (( Busy[N, A ] or Busy[N, B ] ) implies
    (ex N' being Network st
      ( (N' = N) &
      (for X, Y being Station holds
        (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ]) ) &
      (for Z being Station holds
        (Busy[N, Z ] iff Busy[N', Z ]) ) ) ))
  by direct(_PREVIOUS);
  {<<primitives$R272:primitives$R267>>}

end;
{<<primitives$R273:(LAMBDA (primitives$R248) (LAMBDA (primitives$R249)
 (LAMBDA (primitives$R250) primitives$R267)))>>}

NET_Busy_connect: (for N being Network holds
  (for A, B being Station holds
    (( Busy[N, A ] or Busy[N, B ] ) implies
      (ex N' being Network st
        ( (N' = N) &
```

```
        (for X, Y being Station holds
          (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ]) ) &
        (for Z being Station holds
          (Busy[N, Z ] iff Busy[N', Z ]) ) ) )) ) )
by direct(_PREVIOUS);
{<<primitives.NET_Busy_connect$P:primitives$R273>>}
```

B.6 Proof of NET_Disconnect

```
Disconnect: now
  let N be Network;
  {<<primitives$R213:primitives$R213>> }

  let A, B be Station;
  {<<primitives$R214:primitives$R214>> <<primitives$R215:primitives$R215>> }

  now
    assume 10: IsConnected[N, A, B ];
    {<<primitives$R216:primitives$R216>>}

    /* Since stations A and B are connected in network N, we can state*/
    /* the characteristics of the disconnect operation.               */

    20: ((disconnect <N, A, B >) = (disconnect <N, A, B >))
    by elim[N,A,B](NET_Disconnect_identity, _PREVIOUS);
    30: (not IsConnected[(disconnect <N, A, B >), A, B ])
    by elim[N,A,B](NET_Disconnect_not_connected, 10);
    40: ( (not Busy[(disconnect <N, A, B >), A ]) &
    (not Busy[(disconnect <N, A, B >) , B ]) )
    by elim[N,A,B](NET_Disconnect_not_busy, 10);
    50: (for X, Y being Station holds
      (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[(disconnect <N, A, B >) , X, Y ])) )
    by elim[N,A,B](NET_Disconnect_connections, 10);
    60: (for Z being Station holds
      (( (Z <> A) & (Z <> B) ) implies
        (Busy[N, Z ] iff Busy[(disconnect <N, A, B >), Z ])) )
    by elim[N,A,B](NET_Disconnect_remains_busy, 10);
    ( ((disconnect <N, A, B >) = (disconnect <N, A, B >)) &
    (not IsConnected[(disconnect <N, A, B >), A, B ]) &
    (not Busy[(disconnect <N, A, B >), A ]) &
    (not Busy[(disconnect <N, A, B >), B ]) &
    (for X, Y being Station holds
      (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[(disconnect <N, A, B >) , X, Y ])) ) &
```

```
    (for Z being Station holds
      (( (Z <> A) & (Z <> B) ) implies
        (Busy[N, Z ] iff Busy[(disconnect <N, A, B >), Z ])) ) )
    by conj(20, 30, 40, 50, 60);

    /* Introduce a new network which represents the results of the    */
    /* disconnect operation.                                          */

    thus (ex N' being Network st
      ( (N' = (disconnect <N, A, B >)) & (not IsConnected[N', A, B ]) &
      (not Busy[N', A ]) & (not Busy[N', B ]) &
      (for X, Y being Station holds
        (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
      (for Z being Station holds
        (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[N', Z ])) ) ) )
    by exintro(_PREVIOUS);
    {<<primitives$R233:(LAMBDA () (FUNCALL networks-disconnect (R-TUPLE
    primitives$R213 primitives$R214 primitives$R215))) >>}

  end;
  {<<primitives$R234:(LAMBDA () (FUNCALL networks-disconnect (R-TUPLE
  primitives$R213 primitives$R214 primitives$R215))) >>}

  thus (IsConnected[N, A, B ] implies
    (ex N' being Network st
      ( (N' = (disconnect <N, A, B >)) & (not IsConnected[N', A, B ]) &
      (not Busy[N', A ]) & (not Busy[N', B ]) &
      (for X, Y being Station holds
        (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
      (for Z being Station holds
        (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[N', Z ])) ) ) ))
  by direct(_PREVIOUS);
  {<<primitives$R239:primitives$R234>>}

end;
{<<primitives$R240:(LAMBDA (primitives$R213) (LAMBDA (primitives$R214)
(LAMBDA (primitives$R215) primitives$R234)))>>}

NET_Disconnect: (for N being Network holds
  (for A, B being Station holds
    (IsConnected[N, A, B ] implies
      (ex N' being Network st
```

```
        ( (N' = (disconnect <N, A, B >)) & (not IsConnected[N', A, B ]) &
        (not Busy[N', A ]) & (not Busy[N', B ]) &
        (for X, Y being Station holds
          (( (X <> A) & (X <> B) & (Y <> A) & (Y <> B) ) implies
            (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) &
        (for Z being Station holds
          (( (Z <> A) & (Z <> B) ) implies
          (Busy[N, Z ] iff Busy[N', Z ])) ) ) )) ) )
by direct(_PREVIOUS);
{<<primitives.NET_Disconnect$P:primitives$R240>>}
```

Appendix C

Proof of Forced Connection

Due to the extreme length of the complete proof listing, a pseudo-proof listing is provided. The reason the actual proof is so long is that areas such as proving the existence of the affected network and proving "consistency" are tedious. Although these areas are not long by themselves, they are duplicated numerous times throughout the complete listing. This pseudo-proof listing provides enough of the detail to understand that the forced connection proof was intended to assist in the verification of the specifications of the numerous operations.

One of the interesting parts of how this proof is resolved occurs in the development of the affected network. The following lines of pseudo-proof reflect the steps needed to prove the existence of an affected network.

1. prove there exists an empty network (affected network);
2. prove that a station involved in the operation can be added to the affected network;
3. prove that consistency holds on the original network with the exception of the stations in the affected network;
4. repeat the last two steps until all stations involved in the operation have been added;

These lines will be replaced in the following pseudo-proof by indicating that the affected network is proven. This simplifies the pseudo-proof for ease of reading.

C.1 Pseudo-Proof of Forced Connection

```
Forced_Connection: now
  let N be Network;
  let S1, S2 be Station;

  /* We need to prove the theorem for both cases of whether or not   */
```

```
/* S1 is contained in N.                                          */

now
  assume ContainedIn[N, S1 ];

  /* We need to prove the theorem for both cases of whether or not   */
  /* S2 is contained in N.                                          */

  now
    assume ContainedIn[N, S2];

    /* We need to prove the theorem for both cases of whether or not   */
    /* S1 and S2 are connected in N.                                  */

    now
      assume IsConnected[N, S1, S2];
      prove that a copy of the original network exists;
      prove that the affected network is empty;
      prove the theorem;
    end; /* (IsConnected[N, S1, S2]) */

    now
      assume (not IsConnected[N, S1, S2]);

      /* We need to prove the theorem for both cases of whether or not   */
      /* S1 and S2 are the same station.                                */

      now
        assume (S1 = S2);
        prove that a copy of the original network exists;
        prove that the affected network is empty;
        prove the theorem;
      end; /* (S1 = S2) */

      now
        assume (S1 <> S2);

        /* We need to prove the theorem for both cases of whether or not   */
        /* S1 is busy in N.                                               */

        now
          assume Busy[N, S1];
          prove that the station, B1, connected to S1 can be determined;
          prove that S1 and B1 can be disconnected;

          /* We need to prove the theorem for both cases of whether or not   */
```

```
  /* S2 is busy in N.                                                    */

  now
    assume Busy[N, S2];
    prove that the station, B2, connected to S2 can be determined;
    prove that S2 and B2 can be disconnected;
    prove that S1 and S2 can be connected;
    prove that the affected network consists of S1, S2, B1 and B2;
    prove the theorem;

  end; /* (Busy[N, S2]) */

  now
    assume (not Busy[N, S2]);
    prove that S1 and S2 can be connected;
    prove that the affected network consists of S1, S2 and B1;
    prove the theorem;

  end; /* (not Busy[N, S2]) */

end; /* (Busy[N, S1]) */

now
  assume (not Busy[N, S1]);
  /* We need to prove the theorem for both cases of whether or not    */
  /* S2 is busy in N.                                                 */

  now
    assume Busy[N, S2];
    prove that the station, B2, connected to S2 can be determined;
    prove that S2 and B2 can be disconnected;
    prove that S1 and S2 can be connected;
    prove that the affected network consists of S1, S2 and B2;
    prove the theorem;

  end; /* (Busy[N, S2]) */

  now
    assume (not Busy[N, S2]);
    prove that S1 and S2 can be connected;
    prove that the affected network consists of S1 and S2;
    prove the theorem;

  end; /* (not Busy[N, S2]) */

end; /* (not Busy[N, S1]) */
```

```
        end; /* (S1 <> S2) */

      end; /* (not IsConnected[N, S1, S2]) */

    end; /* (ContainedIn[N, S2]) */

    now
      assume (not Contained[N, S2]);
      prove that a copy of the original network exists;
      prove that the affected network is empty;
      prove the theorem;
    end; /* (not Contained[N, S2]) */

  end; /* (Contained[N, S1]) */

  now
    assume (not Contained[N, S1];
    prove that a copy of the original network exists;
    prove that the affected network is empty;
    prove the theorem;
  end; /* (not Contained[N, S1]) */

  /* Since we have proven the theorem for all cases we can conclude  */
  /* the theorem.                                                    */

  thus (ex N' being Network st
    ( ( IsConnected[N', S1, S2 ] or (N' = N) ) &
    (ex Affected being Network st
      (for X, Y being Station holds
        (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) ) ) );
end; /* Forced_Connection */

/* With the completion of the proof, we can conclude the theorem for  */
/* the forced connection for all stations S1 and S2 and network N.    */

forced_connection: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2 ] or (N' = N) ) &
      (ex Affected being Network st
        (for X, Y being Station holds
          (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
            (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])) ) ) ) ) );
```

Appendix D

Proof of Basic Call Attempt

The complete proof listing for the basic call attempt theorem is extremely long. In order to ease readability, a pseudo-proof listing is provided. The areas which prove the existence of an affected network and "consistency" are similar to the steps identified in the introduction of the forced connection proof in appendix C. The only difference is that the identification of the stations added to the affected network is maintained in order to express them in the theorem.

The reader will find that the structure of the call attempt proof is similar to that of the forced connection proof. The only difference is the way the cases in which the stations S1 and S2 are busy are handled. In the forced connection proof, S1 and S2 were disconnected from the current connection they were involved with. This allowed a connection to be created between S1 and S2. In the call attempt proof, if either of the stations S1 or S2 are busy then the original network is returned as the new network.

D.1 Pseudo-Proof of Call Attempt

```
Call_Attempt: now
  let N be Network;
  let S1, S2 be Station;

  /* We need to prove the theorem for both cases of whether or not   */
  /* S1 is contained in N.                                           */

  now
    assume ContainedIn[N, S1 ];

    /* We need to prove the theorem for both cases of whether or not   */
    /* S2 is contained in N.                                           */

    now
      assume ContainedIn[N, S2];
```

```
/* We need to prove the theorem for both cases of whether or not   */
/* S1 and S2 are connected in N.                                    */

now
  assume IsConnected[N, S1, S2];
  prove that a copy of the original network exists;
  prove that the affected network is empty;
  prove the theorem;
end; /* (IsConnected[N, S1, S2]) */

now
  assume (not IsConnected[N, S1, S2]);

  /* We need to prove the theorem for both cases of whether or not   */
  /* S1 and S2 are the same station.                                  */

  now
    assume (S1 = S2);
    prove that a copy of the original network exists;
    prove that the affected network is empty;
    prove the theorem;
  end; /* (S1 = S2) */

  now
    assume (S1 <> S2);

    /* We need to prove the theorem for both cases of whether or not   */
    /* S1 is busy in N.                                                 */

    now
      assume Busy[N, S1];
      prove that a copy of the original network exists;
      prove that the affected network is empty;
      prove the theorem;
    end; /* (Busy[N, S1]) */

    now
      assume (not Busy[N, S1]);
      /* We need to prove the theorem for both cases of whether or not   */
      /* S2 is busy in N.                                                 */

      now
        assume Busy[N, S2];
        prove that a copy of the original network exists;
        prove that the affected network is empty;
```

```
            prove the theorem;
          end; /* (Busy[N, S2]) */

          now
            assume (not Busy[N, S2]);
            prove that S1 and S2 can be connected;
            prove that the affected network consists of S1 and S2;
            prove the theorem;

          end; /* (not Busy[N, S2]) */

        end; /* (not Busy[N, S1]) */

      end; /* (S1 <> S2) */

    end; /* (not IsConnected[N, S1, S2]) */

  end; /* (ContainedIn[N, S2]) */

  now
    assume (not Contained[N, S2]);
    prove that a copy of the original network exists;
    prove that the affected network is empty;
    prove the theorem;
  end; /* (not Contained[N, S2]) */

end; /* (Contained[N, S1]) */

now
  assume (not Contained[N, S1];
  prove that a copy of the original network exists;
  prove that the affected network is empty;
  prove the theorem;
end; /* (not Contained[N, S1]) */

/* Since we have proven the theorem for all cases we can conclude  */
/* the theorem.                                                    */

thus (ex N' being Network st
  ( ( IsConnected[N', S1, S2 ] or (N' = N) ) &
  (ex Affected being Network st
    ( (for X, Y being Station holds
      (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
        (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ])))) &
    (for Z being Station holds
      (ContainedIn[Affected, Z ] implies
```

```
        ( (Z = S1) or (Z = S2) )) ) ) ) ) );

end; /* Call_Attempt */

/* With the completion of the proof, we can conclude the theorem for  */
/* the basic call attempt for all stations S1 and S2 and network N.   */

basic_call: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2 ] or (N' = N) ) &
      (ex Affected being Network st
        ( (for X, Y being Station holds
          (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
            (IsConnected[N, X, Y ] iff IsConnected[N', X, Y ]))) &
        (for Z being Station holds
          (ContainedIn[Affected, Z ] implies
            ( (Z = S1) or (Z = S2) )) ) ) ) ) ) ) );
```

Appendix E

Feature Formalization

E.1 Proof of Call Display

```
display.Call_Display: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N', S2] ) implies
        (ex Id being Identifier st
          (Id=(get_id <N', S1>)))) ))))
rby display.Call_Display$P;


call_display: now
  let N be Network;
  {<<display$R1062:display$R1062>> }

  let S1, S2 be Station;
  {<<display$R1063:display$R1063>> <<display$R1064:display$R1064>> }

  /* Introduce the theorem regarding a call attempt between stations */
  /* S1 and S2 in network N.                                         */

  consider N1 being Network such that
    ( ( IsConnected[N1, S1, S2 ] or (N1 = N) ) &
    (ex Affected being Network st
      (for X, Y being Station holds
        (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[N1, X, Y ])) ) ) )
  by elim[N,S1,S2](basic.basic_call);
  {<<display$R1065:(R-APPLY (R-APPLY (R-APPLY basic.basic_call$P
  display$R1062 :tag '$ELIM21074) display$R1063 :tag '$ELIM21075)
  display$R1064 :tag '$ELIM21076)>>}
```

```
{<<display$R1073:(R-APPLY0 (R-FIRST display$R1065) :tag '$EXISTCO1072)>> }
{<<display$R1071:(R-APPLY0 (R-SECOND display$R1065) :tag '$EXISTCF1070)>>}
call_attempt: basic.BasicCall[N, S1, S2, N1 ] by definition(basic.BasicCall, _PREVIOUS);
{<<display$R1077:display$R1071>>}
now
  assume ( IsConnected[N1, S1, S2 ] & DisplayOn[N1, S2 ] );
  {<<display$R1078:display$R1078>>}

  /* Since we assume S1 and S2 are connected in N and S2 has call     */
  /* display on, we can identify the identifier of S1.                */

  ((get_id <N1, S1 >) = (get_id <N1, S1 >))
  by elim[N1,S1](NET_Get_identifier);
  thus (ex Id being Identifier st
    (Id = (get_id <N1, S1 >)) )
  by exintro(_PREVIOUS);
  {<<display$R1081:(LAMBDA () (FUNCALL networks-get_id
  (R-TUPLE display$R1073 display$R1063))) >>}

end;
{<<display$R1082:(LAMBDA () (FUNCALL networks-get_id
(R-TUPLE display$R1073 display$R1063))) >>}

(( IsConnected[N1, S1, S2 ] & DisplayOn[N1, S2 ] ) implies
  (ex Id being Identifier st
    (Id = (get_id <N1, S1 >)) ))
by direct(_PREVIOUS);
{<<display$R1084:display$R1082>>}
( basic.BasicCall[N, S1, S2, N1 ] &
(( IsConnected[N1, S1, S2 ] & DisplayOn[N1, S2 ] ) implies
  (ex Id being Identifier st
    (Id = (get_id <N1, S1 >)) )) )
by conj(call_attempt, _PREVIOUS);
{<<display$R1086:(R-LIST  display$R1077 display$R1084)>>}

/* Introduce the new network which represents the results of the    */
/* call attempt and the side effects of having call display on.     */

thus (ex N' being Network st
  ( basic.BasicCall[N, S1, S2, N' ] &
  (( IsConnected[N', S1, S2 ] & DisplayOn[N', S2 ] ) implies
    (ex Id being Identifier st
      (Id = (get_id <N', S1 >)) )) ) )
by exintro(_PREVIOUS);
{<<display$R1089:(R-LIST (LAMBDA () display$R1073) (LAMBDA ()
display$R1086))>>}
```

```
end;
{<<display$R1090:(LAMBDA (display$R1062) (LAMBDA (display$R1063) (LAMBDA
(display$R1064) (R-LIST (LAMBDA () display$R1073) (LAMBDA ()
display$R1086)))))>>}

Call_Display: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N' ] &
      (( IsConnected[N', S1, S2 ] & DisplayOn[N', S2 ] ) implies
        (ex Id being Identifier st
          (Id = (get_id <N', S1 >)) )) ) ) ) )
by direct(_PREVIOUS);
{<<display.Call_Display$P:display$R1090>>}
```

E.2 Proof of Call Block

```
block.Call_Block: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N, S2] & BlockOn[N, S1] ) implies
        (ex Id being Identifier st
          (Id=UnknownId))) ))))
rby block.Call_Block$P;

call_block: now
  let N be Network;
  {<<block$R158:block$R158>>}

  let S1, S2 be Station;
  {<<block$R159:block$R159>> <<block$R160:block$R160>>}

  /* Introduce the theorem regarding a call attempt between stations */
  /* S1 and S2 in network N.                                         */

  consider N1 being Network such that
    ( ( IsConnected[N1, S1, S2 ] or (N1 = N) ) &
    (ex Affected being Network st
      (for X, Y being Station holds
        (( (not ContainedIn[Affected, X ]) & (not ContainedIn[Affected, Y ]) ) implies
          (IsConnected[N, X, Y ] iff IsConnected[N1, X, Y ])) ) ) )
  by elim[N,S1,S2](basic.basic_call);
  {<<block$R161:(R-APPLY (R-APPLY (R-APPLY basic.basic_call$P
  block$R158 :tag '$ELIM2170) block$R159 :tag '$ELIM2171) block$R160
```

```
:tag '$ELIM2172)>>}
{<<block$R169:(R-APPLY0 (R-FIRST block$R161) :tag '$EXISTC0168)>>}
{<<block$R167:(R-APPLY0 (R-SECOND block$R161) :tag '$EXISTCF166)>>}
call_attempt: basic.BasicCall[N, S1, S2, N1 ] by definition(basic.BasicCall, _PREVIOUS);
{<<block$R173:block$R167>>}
now
  assume ( IsConnected[N1, S1, S2 ] & DisplayOn[N, S2 ] & BlockOn[N, S1 ] );
  {<<block$R178:block$R178>>}

  /* Since we assume S1 and S2 are connected in N, S2 has call       */
  /* display on, and S1 has call block on, we can identify the       */
  /* identifier of S1 as being the unknown identifier.               */

  BlockOn[N, S1 ] by conj(_PREVIOUS);
  ((get_id <N, S1 >) = UnknownId)
  by elim[N,S1](NET_get_id_block_on, _PREVIOUS);
  thus (ex Id being Identifier st
    (Id = UnknownId) )
  by exintro(_PREVIOUS);
  {<<block$R182:(LAMBDA () (FUNCALL networks-get_id (R-TUPLE block$R158
  block$R159))) >>}

end;
{<<block$R183:(LAMBDA () (FUNCALL networks-get_id (R-TUPLE block$R158
block$R159))) >>}

(( IsConnected[N1, S1, S2 ] & DisplayOn[N, S2 ] & BlockOn[N, S1 ] ) implies
  (ex Id being Identifier st
    (Id = UnknownId) ))
by direct(_PREVIOUS);
{<<block$R185:block$R183>>}
( basic.BasicCall[N, S1, S2, N1 ] &
(( IsConnected[N1, S1, S2 ] & DisplayOn[N, S2 ] & BlockOn[N, S1 ] ) implies
  (ex Id being Identifier st
    (Id = UnknownId) )) )
by conj(call_attempt, _PREVIOUS);
{<<block$R187:(R-LIST  block$R173 block$R185)>>}

/* Introduce the new network which represents the results of the    */
/* call attempt and the side effects of having call display on and  */
/* call block on for stations involved in the connection.          */

thus (ex N' being Network st
  ( basic.BasicCall[N, S1, S2, N' ] &
  (( IsConnected[N', S1, S2 ] & DisplayOn[N, S2 ] & BlockOn[N, S1 ] ) implies
    (ex Id being Identifier st (Id = UnknownId) )) ) )
```

```
  by exintro(_PREVIOUS);
  {<<block$R190:(R-LIST (LAMBDA () block$R169) (LAMBDA () block$R187))>>}

end; {<<block$R197:(LAMBDA (block$R158) (LAMBDA (block$R159) (LAMBDA
(block$R160) (R-LIST (LAMBDA () block$R169) (LAMBDA ()
block$R187))))))>>}

Call_Block: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N' ] &
      (( IsConnected[N', S1, S2 ] & DisplayOn[N, S2 ] & BlockOn[N, S1 ] ) implies
        (ex Id being Identifier st
          (Id = UnknownId) )) ) ) ) )
by direct(_PREVIOUS);
{<<block.Call_Block$P:block$R197>>}
```

Appendix F

Executions of Extracted Code

The following details the execution of the xlisp code which was extracted from the proofs of the various features and operations. For further information regarding the production and extraction of code from proofs in MIZAR-C please refer to Kippen [14] and Ho [13].

For each of these examples, the following format is used to describe a network using lists in xlisp:

```
(setq N1 '( (((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((A 123) (D 456))
  ((A 123) (B 234)))))
```

This defines a variable, *N1*, which represents a network. This list contains four sublists. The first lists all the stations in the network in a format of *(Name Identifier)*. The second sublist lists all the connections that currently exist in the network. It is in the format of *(Station Station)*. Stations which have the call display feature on in the network are listed in the third sublist and the fourth contains a list of the stations with call block on.

F.1 Forced Connection

The following is the theorem defining the forced connection operation:

```
forced.forced_connection: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2] or (N'=N) ) & (ex Affected being Network st
        (for X, Y being Station holds
          (( (not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y]) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y]))))) ))))
rby forced.forced_connection$P;
```

After extraction of the code, xlisp was used to execute the function and verify the results. The first step was to define the function that we would use. The defun statement defines the function forced_connection which takes three arguments, *N*, *S1* and *S2*. These represent the network and the two stations involved in the operation.

```
(defun forced_connection (N S1 S2) (r-apply (r-apply (r-apply (first Theorem) N) S1) S2))
```

The arguments, *N*, *S1*, and *S2*, were then declared. Here we have introduced a network containing six stations, *A* to *F*, two connections, *A* to *B* and *C* to *D*, one station, *D*, with call display enabled and no stations with call block enabled. The stations that we want to force a connection between are *D* and *B*.

```
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((D 456)) ()))
(setq S1 '(D 456) )
(setq S2 '(B 234) )
```

The result is then defined as the application of the *forced_connection* function on network *N1* and stations *S1* and *S2*.

```
(setq result (forced_connection N1 S1 S2))
```

From the result, we can extract the information that can be determined as highlighted in the theorem of *forced_connection*. First, we can extract the resulting network:

```
(r-apply0 (first result))
```

The resulting network is the following in which stations *D* and *B* are connected and, in the process, the other two connections are eliminated.

```
(((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((D 456) (B 234)))
  ((D 456))
  ())
```

We are also able to extract the resulting affected network to determine which stations were involved in the operation. The following extracts the affected network from the result of the *forced_connection* function:

```
(r-apply0 (second (r-apply0 (second result))))
```

The affected network contains stations *A* and *B* since they had to be disconnected to make station *B* not busy. Stations *C* and *D* where also disconnected but to make station *D* not busy.

```
(((D 456) (B 234) (C 345) (A 123)) () () ())
```

F.2 Basic Call Attempt

The following is the theorem defining the basic call attempt operation:

```
basic.basic_call: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( ( IsConnected[N', S1, S2] or (N'=N) ) &
      (ex Affected being Network st
        (for X, Y being Station holds
          (( (not ContainedIn[Affected, X]) & (not ContainedIn[Affected, Y]) ) implies
            (IsConnected[N, X, Y] iff IsConnected[N', X, Y])) &
        (for Z being Station holds
          (ContainedIn[Affected, Z] implies ( (Z=S1) or (Z=S2) )))) ))))
rby basic.basic_call$P;
```

First, the function *basic_call* was defined in xlisp:

```
(defun basic_call (N S1 S2) (r-apply (r-apply (r-apply (first Theorem) N) S1) S2))
```

As a first example of the *basic_call* function, *N1*, *S1*, and *S2* where defined so that *S1* and *S2* were both not busy in the network.

```
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((D 456))
  ()))
(setq S1 '(E 567) )
(setq S2 '(F 678) )
```

The result of applying the *basic_call* function to *N1*, *S1*, and *S2* was defined:

```
(setq result (basic_call n1 s1 s2))
```

The resulting network was one in which all connections were maintained and the new connection was made:

```
(r-apply0 (first result))
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)) ((E 567)(F 678)))
  ((D 456))
  ()))
```

The affected network formed by the operation contained only stations E and F since they were the only ones involved in the operation:

```
(r-apply0 (second (r-apply0 (second result))))
(((E 567) (F 678)) () () ())
```

As a second example, *N1*, *S1*, and *S2* were defined so that one of the stations would be currently busy.

```
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((D 456))
  ()))
  (setq S1 '(A 123) )
  (setq S2 '(F 678) )
```

Again the result was defined as apply the *basic_call* function to *N1*, *S1*, and *S2*.

```
(setq result (basic_call N1 S1 S2))
```

This resulted in the original network being returned since the connection could not be made:

```
(r-apply0 (first result))
(((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((D 456))
  ())
```

Furthermore, the resulting affected network was empty since no stations were affected in the operation:

```
(r-apply0 (second (r-apply0 (second result))))
(() () () ())
```

F.3 Call Display

The following is the theorem defining the effects of call display feature during a call attempt:

```
display.Call_Display: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N', S2] ) implies
        (ex Id being Identifier st
          (Id=(get_id <N', S1>)))) ))))
  rby display.Call_Display$P;
```

First, the function *call_display* was defined in xlisp:

```
(defun call_display (N S1 S2) (r-apply (r-apply (r-apply (first Theorem) N) S1) S2))
```

N1, *S1*, and *S2* where defined so that *S1* and *S2* were both not busy in the network and *S2* had call display enabled.

```
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((A 123)(E 567))
  ()) )
(setq S1 '(F 678) )
(setq S2 '(E 567) )
```

The result of applying the *call_display* function to *N1*, *S1*, and *S2* was defined:

```
(setq result (call_display n1 s1 s2))
```

The resulting network was one in which all connections were maintained and the new connection was made:

```
(r-apply0 (first result))
(((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)) ((F 678) (E 567)))
  ((A 123)(E 567))
  ())
```

The affected network formed by the operation contained only stations E and F since they were the only ones involved in the operation:

```
(r-apply0 (second (r-apply0 (second result))))
(((E 567) (F 678)) () () ())
```

Furthermore, we are able to find the identifier of the calling station since the called station had call display enabled:

```
(r-apply0 (second (r-apply0 (second result))))
678
```

F.4 Call Block

The following is the theorem defining the effects of call block feature during a call attempt:

```
block.Call_Block: (for N being Network holds
  (for S1, S2 being Station holds
    (ex N' being Network st
      ( basic.BasicCall[N, S1, S2, N'] &
      (( IsConnected[N', S1, S2] & DisplayOn[N, S2] & BlockOn[N, S1] ) implies
        (ex Id being Identifier st
          (Id=UnknownId))) ))))
rby block.Call_Block$P;
```

First, the function *call_block* was defined in xlisp:

```
(defun call_block (N S1 S2) (r-apply (r-apply (r-apply (first Theorem) N) S1) S2))
```

N1, *S1*, and *S2* where defined so that *S1* and *S2* were both not busy in the network, *S2* had call display enabled, and *S1* had call block enabled.

```
(setq N1 '( ((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)))
  ((A 123)(E 567))
  ((F 678))) )
(setq S1 '(F 678) )
(setq S2 '(E 567) )
```

The result of applying the *call_block* function to *N1*, *S1*, and *S2* was defined:

```
(setq result (call_block n1 s1 s2))
```

The resulting network was one in which all connections were maintained and the new connection was made:

```
(r-apply0 (first result))
(((A 123) (B 234) (C 345) (D 456) (E 567) (F 678))
  (((A 123) (B 234)) ((C 345) (D 456)) ((F 678) (E 567)))
  ((A 123)(E 567))
  ())
```

The affected network formed by the operation contained only stations *E* and *F* since they were the only ones involved in the operation:

```
(r-apply0 (second (r-apply0 (second result))))
(((E 567) (F 678)) () () ())
```

Furthermore, we are able to find that the identifier of the calling station was the unknown identifier since the called station had call display enabled and the calling station had call block enabled:

```
(r-apply0 (second (r-apply0 (second result))))
"UnknownId"
```

Bibliography

[1] S. Berardi. Pruning simply typed lambda-terms. Technical report, Dipartimento di Informatica dell'Universita' di Torino (University of Turin, Italy).

[2] Ewa Bonarska. *An Introduction to PC Mizar.* Mizar Users Group. Fondation Philippe le Hodey, Brussels, 1990.

[3] Rezki Boumezbeur and Luigi Logrippo. Specifying telephone systems in lotos. *IEEE Communications*, 31(8):38–45, August 1993. 1^{st} International Workshop on Feature Interactions in Telecommunications Systems.

[4] T. F. Bowen et al. Views on the feature-interaction problem. *Proceedings of the Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, pages 59–62, 1989.

[5] Ricky Butler. An elementary tutorial on formal specification and verification using pvs 2. Technical Report 108991, National Aeronautics and Space Administration, Langley Research Center, 1993.

[6] Cameron et al. A feature-interaction benchmark for in and beyond. *IEEE Communications*, 31(3), March 1993. Shortened Version.

[7] K. E. Cheng. Towards a formal model for incremental service specification and interaction management support. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, Washington, DC, 1994. IOS Press. 2^{nd} International Workshop on Feature Interactions in Telecommunications Systems.

[8] P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, Washington, DC, 1994. IOS Press. 2^{nd} International Workshop on Feature Interactions in Telecommunications Systems.

[9] M. Faci and L. Logrippo. Specifying features and analysing their interactions in a lotos environment. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, Washington, DC, 1994. IOS Press. 2^{nd} International Workshop on Feature Interactions in Telecommunications Systems.

[10] Alan Fekete. Formal models of communication services: A case study. *IEEE Computer*, 26(3):37–47, August 1993. 1st International Workshop on Feature Interactions in Telecommunications Systems.

[11] Nancy D. Griffeth and Yow-Jian Lin. Extending telecommunications systems: The feature interaction problem. *IEEE Computer*, 26(3):14–18, August 1993. 1st International Workshop on Feature Interactions in Telecommunications Systems.

[12] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between vdm and z. Technical Report UMCS-93-8-1, University of Manchester, 1993.

[13] Edward Ho. Implementing binary trees in mizar-c. Master's thesis, University of Alberta, 1996.

[14] Kathy Kippen. Implementing bit data structures in mizar-c. Master's thesis, University of Alberta, 1995.

[15] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[16] B. Knight. Safe strict evaluation of redundancy-free programs from proofs. Master's thesis, University of Victoria, 1994.

[17] Michal Muzalewski. *An Outline of PC Mizar*. Mizar Users Group. Fondation Philippe le Hodey, Brussels, 1993.

[18] K. Prażmowski, P. Rudnicki, et al. Mizar-MSE Primer and User Guide. TR 88-9, The University of Alberta, Department of Computing Science, Edmonton, 1988.

[19] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1989.

[20] Hugo Velthuijsen. Distributed artificial intelligence for runtime feature-interaction resolution. *IEEE Computer*, 26(3):48–55, August 1993. 1st International Workshop on Feature Interactions in Telecommunications Systems.

[21] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(3):20–29, August 1993. 1st International Workshop on Feature Interactions in Telecommunications Systems.